MacTech Magazine
Vol. 14, No. 12 • December 1998

For Macintosh Programmers & Developers

MacTech®

INSIDE: PowerPlant Fundamentals!

OpenGL

on the M

by Ed Ang

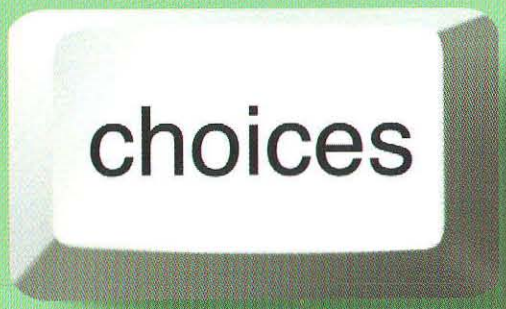
Also Inside:

Natural Rotations: the right way to do 3D rotations
by Kas Thomas

Delayed Messaging in Yellow Box
by Mike Morton

0 32128 74887 8 12

$8.95 US
$9.95 Canada
ISSN 1067-8360
Printed in U.S.A.



The industry's only provider of ADB, USB and software-based license management.

It's all about choices and at Rainbow Technologies we give Macintosh software developers more. As the industry leader, Rainbow offers the most secure, reliable and technologically advanced solutions to piracy and increased sales.

SentinelEve3-USB
All the technology and reliability of the SentinelEve3 in a USB model

SentinelEve3
The leading ADB-based protection solution for Macintosh developers

SentinelEve3 — ADB-based protection
SentinelEve3-USB — ready for the new iMac
SentinelLM-Mac — a software-based license manager for standalone and network applications including Java

SentinelLM
comprehensive license management for standalone or networks

To learn more about the choices for secure licensing, call now or visit our website www.rainbow.com/macintosh. You'll see that for Macintosh developer solutions, there's only one choice — Rainbow.

RAINBOW TECHNOLOGIES

Changing the way the world secures business.

50 Technology Drive, Irvine, CA 92618
800.705.5552 tel. 949.450.7300 fax. 949.450.7450 www.rainbow.com/macintosh
Offices and distributors located worldwide.

"Without a doubt, the Premiere Resource Editor for the Mac OS ... A wealth of time-saving tools."
– MacUser Magazine Eddy Awards

"A distinct improvement over Apple's ResEdit."
– MacTech Magazine

"Every Mac OS developer should own a copy of Resorcerer."
– Leonard Rosenthol, Aladdin Systems

"Without Resorcerer, our localization efforts would look like a Tower of Babel. Don't do product without it!"
– Greg Galanos, CEO and President, Metrowerks

"Resorcerer's data template system is amazing."
– Bill Goodman, author of Smaller Installer and Compact Pro

"Resorcerer Rocks! Buy it, you will NOT regret it."
– Joe Zobkiw, author of ***A Fragment of Your Imagination***

"Resorcerer will pay for itself many times over in saved time and effort."
– MacUser review

"The template that disassembles 'PICT's is awesome!"
– Bill Steinberg, author of Pyro! and PBTools

"Resorcerer proved indispensible in its own creation!"
– Doug McKenna, author of Resorcerer

Resorcerer® 2

Version 2.0

The Resource Editor for the Mac™ OS Wizard

ORDERING INFO

Requires System 7.0 or greater,
1.5MB RAM, CD-ROM

Standard price: $256 (decimal)
Website price: $128 - $256
(Educational, quantity, or
other discounts available)

Includes: Electronic documentation
60-day Money-Back Guarantee
Domestic standard shipping

Payment: Check, PO's, or Visa/MC
Taxes: Colorado customers only

Extras (call, fax, or email us):
COD, FedEx, UPS Blue/Red,
International Shipping

MATHEMÆSTHETICS, INC.
PO Box 298
Boulder, CO 80306-0298 USA
Phone: (303) 440-0707
Fax: (303) 440-0504
resorcerer@mathemaesthetics.com

New in 2.0:

- **Very fast, HFS browser for viewing file tree of all volumes**
- **Extensibility for new Resorcerer Apprentices (CFM plug-ins)**
- **New AppleScript Dictionary ('aete') Apprentice Editor**
- **MacOS 8 Appearance Manager-savvy Control Editor**
- **PowerPlant text traits and menu command support**
- **Complete AIFF sound file disassembly template**
- **Big-, little-, and even mixed-endian template parsing**
- **Auto-backup during file saves; folder attribute editing**
- **Ships with PowerPC native, fat, and 68K versions**

- Fully supported; it's easier, faster, and more productive than ResEdit
- Safer memory-based, not disk-file-based, design and operation
- All file information and common commands in one easy-to-use window
- Compares resource files, and even **edits your data forks** as well
- Visible, accumulating, editable scrap
- Searches and opens/marks/selects resources by text content
- Makes global resource ID or type changes easily and safely
- Builds resource files from simple Rez-like scripts
- Most editors DeRez directly to the clipboard
- All graphic editors support screen-copying or partial screen-copying
- Hot-linking Value Converter for editing 32 bits in a dozen formats
- Its own 32-bit List Mgr can open and edit very large data structures
- Templates can pre- and post-process any arbitrary data structure
- Includes nearly 200 templates for common system resources
- TMPLs for Installer, MacApp, QT, Balloons, AppleEvent, GX, etc.
- Full integrated support for editing color dialogs and menus
- Try out balloons, 'ictb's, lists and popups, even create C source code
- Integrated single-window Hex/Code Editor, with patching, searching
- Editors for cursors, versions, pictures, bundles, and lots more
- Relied on by thousands of Macintosh developers around the world

To order by credit card, or to get the latest news, bug fixes, updates, and apprentices, visit our website...

www.mathemaesthetics.com

For Macintosh
Programmers & Developers

MacTech®

A publication of XPLAIN CORPORATION

How To Communicate With Us

In this electronic age, the art of communication has become both easier and more complicated. Is it any surprise that we prefer **e-mail?**

If you have any questions, feel free to call us at 805/494-9797 or fax us at 805/494-9798.

If you would like a subscription or need customer service, feel free to contact Developer Depot Customer Service at 800-MACDEV-1

DEPARTMENTS	E-Mail/URL
Orders, Circulation, & Customer Service	cust_service@mactech.com
Press Releases	press_releases@mactech.com
Ad Sales	ad_sales@mactech.com
Editorial	editorial@mactech.com
Programmer's Challenge	prog_challenge@mactech.com
Online Support	online@mactech.com
Accounting	accounting@mactech.com
Marketing	marketing@mactech.com
General	info@mactech.com
Web Site (articles, info, URLs and more...)	http://www.mactech.com

MACTECH MAGAZINE

MacTech Magazine is grateful to the following individuals who contribute on a regular basis. We encourage others to share the technology.

We are dedicated to the distribution of useful programming information without regard to Apple's developer status.

*For information on submitting articles, ask us for our **writer's kit** which includes the terms and conditions upon which we publish articles.*

Editorial Board of Advisors

Jordan J. Mattson, Jim Straus, and Jon Wiederspan

Editorial Staff

Publisher • Neil Ticktin
Editor Emeritus • Eric Gundrum
Editor • Nick "nick.c" DeMello
Managing Editor • Jessica Courtney
Online Editor • Jeff Clites
Web Editors • Kevin Avila, Scott Anchin

Contributing Editors

- Michael Brian Bentley
- Tantek Çelik, Microsoft Corporation
- Richard Clark
- Marshall Clow
- Carl de Cordova
- Andrew S. Downs
- Jim Gochee, Connectix
- Steve Kiene, Mindvision
- Peter N Lewis
- Gary Little
- Rich Morin
- Ed Ringel
- Michael Rutman
- Rich Siegel, Bare Bones Software, Inc.

Regular Columnists

Getting Started • Dave Mark and Dan Parks Sydow
Programmer's Challenge • Bob Boonstra
From the Factory Floor • Dave Mark, Metrowerks
Tips & Tidbits • Jeff Clites

XPLAIN CORPORATION

Chief Executive Officer • Neil Ticktin
President • Andrea J. Sniderman
Controller • Michael Friedman
Production Manager • Jessica Courtney
Production • Lorin Welander II
Director of Advertising • Dale Hansman
Marketing Manager • Nick DeMello
Events Manager • Susan M. Worley
Network Administrator • Chris Barrus
Accounting • Jan Webber, Marcie Moriarty
Customer Relations • Molly Covin, Lee Ann Pham, Susan Pomrantz

Board of Advisors • Steven Geller, Blake Park, and Alan Carsrud

This publication is printed on paper with recycled content.

MacTech Magazine (ISSN: 1067-8360 / USPS: 010-227) is published monthly by Xplain Corporation, 850-P Hampshire Road, Westlake Village, CA 91361-2800. Voice: 805/494-9797, FAX: 805/494-9798. Domestic subscription rates are $47.00 per year. Canadian subscriptions are $59.00 per year. All other international subscriptions are $97.00 per year. Domestic source code disk subscriptions are $77 per year. All international disk subscriptions are $97.00 a year. Please remit in U.S. funds only. Periodical postage is paid at Thousand Oaks, CA and at additional mailing office.

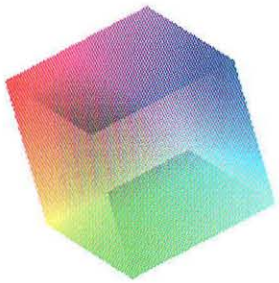
POSTMASTER: Send address changes to **MacTech Magazine**, P.O. Box 5200, Westlake Village, CA 91359-5200.

For Macintosh Programmers & Developers

MacTech®

Contents

November 1998 • Volume 14, No. 11

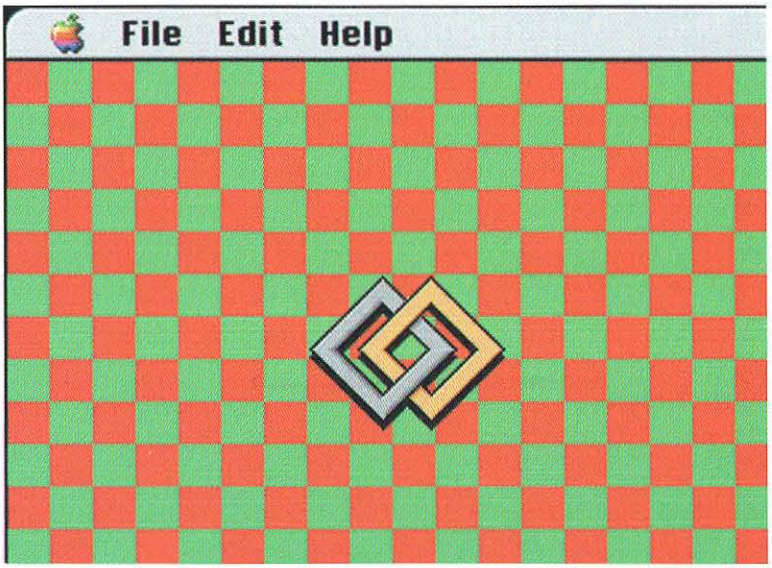


OpenGL for Mac Users..... page 26

Feature Articles

Reader Resources

Columns



Getting Started: Color Animation..... page 6

About the cover...
This month's cover was created by ***Bob Beretta*** *of Conix Enterprises, Inc. <http://www.conixd3d.com> using the Conix OpenGL library and Conix's 3D Explorer — an OpenGL rendering system for Mathematica, by Wolfram Research <http://www.wolfram.com>.*

VIEWPOINT

by Nicholas C. "nick.c" DeMello <editor@mactech.com>

It's been a good year for Apple.

Last year was a hard one. In the second half of 1997, Apple focused on Apple, they made the hard choices, they did a lot of needed house cleaning, and made a lot of necessary sacrifices. This year, Apple focused on Macintosh users, found out what their customers really wanted and they gave it to them. Apple put out two solid and successful system releases (pretty much on schedule). They introduced two innovative and well received new products (the G3 PowerBook and the iMac), and turned out four solid quarters of profitability. This was the turn around year, where Apple worked hard and put down the foundation for a revitalized Macintosh platform.

So, how do you end of a year like 1998? You throw a party.

In less than four weeks, over 70,000 raving, rabid Mac lovers are going to converge on the Moscone center in San Francisco, California to celebrate the Macintosh. If you haven't been to Macworld <http://www.macworldexpo.com/> before, this is the year to start — Mac lovers have a lot to celebrate.

Macworld is a party, a chance to revel with other Mac lovers. But, it's also an opportunity. It's the best chance of the year to find out what people are creating for the Macintosh and how end users are receiving it. It's a chance to explore the software that is growing on the revitalized Mac platform.

At Macworld, marketers and Macintosh end users collide on an unparalleled scale. Each day, the show floor will look like the New York Stock Exchange. Thousands of people will be rushing from one booth to the next, trying to collect and distribute as much information as possible before the bell rings. Conversations will be fast and focused, and slips of business card shaped paper will be passed back and forth (usually with a quick note sketched on the back, to remind us of the conversation details). Each night, the bars and hotel rooms of the city will be packed with people discussing new product releases, negotiating future opportunities, exchanging contacts, and speculating on future directions of the products and the platform.

I didn't mention anything about building the software, did I?

Well, Macworld isn't really about development, it is about what happens after the software leaves our hands. That doesn't mean you and I get the week off though. As a developer, you *need* to be at Macworld to listen to your end users, to make sure that the solutions you're offering are the ones your users want. You need to make sure that you *"get it"*.

Stand near the guy marketing your software and listen to how he sells it — watch which pitches hit home with the users (those are the features that were worth building into your app). Listen to the comments of the customers (the good, and especially the bad). Listen to what is said about your competitor's products, listen to what is said about completely unrelated products. This information about how your products are positioned, about what people think of them, about what people want, is priceless. No where else are you likely to get the quantity, and quality of feedback that is available at Macworld.

Apple, of course, will have the largest section at Macworld. At the last Macworld, the AppleScript section was a hit, it should be a good place to find out what end users are doing with AppleScript (and think about how your applications can offer alternative or cooperative solutions to their problems). It's always interesting to hear how Apple describes fundamental technologies, like MRJ and QuickTime, to end users. Most importantly, make sure you visit the OS section to hear what features of the new Mac OS end users are most impressed with (those are the features that you want to support in your next release).

Look for the *Developer Central* pavilion. *Developer Central* <http://www.devcentral.com/> is a special interest section of Macworld, focused on Macintosh developers and developer products. This is your chance to offer that same priceless feedback to the folks who make the tools you rely on every day. Start building a list now of quirks, bugs, and new features you want to talk to Metrowerks or Mathemaesthtics about. *Developer Central* is also a place to see what new development tools are being released and talk with the authors of these programs to get real time feedback on how these tools can accelerate your development cycle.

Steve Jobs will be delivering the Macworld keynote speech on Tuesday, January 5, at 9:00 a.m. No doubt Jobs will remind us about the Apple successes of 1998. He'll talk about Mac OS 8.5 adoption and iMac sales, but more importantly, he'll talk about 1999 and what Apple's plans are for the new year. I don't know what those plans will be, but, if 1997 was the year Apple focused on Apple, if 1998 was the year Apple concentrated on the Macintosh user, I expect 1999 to be a year that will be especially important to the Macintosh developer.

MacTech will be at the center of *Developer Central*, be sure to stop by and talk to us. We're very interested in hearing what you liked, and didn't like about MacTech in 1998, what articles you want to see in 1999, or just what you thought about this year's keynote speech. See you there!

MacHASP USB

Protects Your Software

Your software is your baby – and you want to look after it. You created it, you developed it, you saw it right through to the moment it was ready for market. Now protect it. 50% of business software is stolen; $11 billion of developers' income is lost to piracy.* Is your software a statistic?

All over the world, more developers are protecting against piracy. They're protecting more products, on more platforms, with more security – and selling more as a result. And more of these developers are protecting with HASP.

To see why 25,000 developers worldwide protect with Aladdin, call us today!

MacHASP USB – So small, So easy, So powerful

- Protects software via any available USB port
- Unequaled security
- Unparalleled flexibility, ease-of-use and transparency
- Compatible with MacHASP keys and software
- Hot swappable
- Contains 100 bytes of memory
- Available in stand-alone and network versions

HASP Protects More

North America	**Aladdin Knowledge Systems Inc.** 800 223-4277, 212 564-5678, Fax: 212 564-3377, Email: hasp.sales@us.aks.com
Int'l Office	**Aladdin Knowledge Systems Ltd.** +972 3 636-2222, Fax: +972 3 537-5796, Email: hasp.sales@aks.com
Germany	**Aladdin Knowledge Systems GmbH & Co. KG** +49 89 89 42 21-0, Fax: +49 89 89 42 21-40, Email: info@aladdin.de
UK	**Aladdin Knowledge Systems UK Ltd.** +44 1753 622266, Fax: +44 1753 622262, Email: sales@aldn.co.uk
Japan	**Aladdin Japan Co., Ltd.** +81 426 60-7191, Fax: +81 426 60-7194, Email: sales@aladdin.co.jp
France	**Aladdin France SA** +33 1 41-37-70-30, Fax: +33 1 41-37-70-39, Email: info@aladdin.fr
Benelux	**Aladdin Software Security Benelux B.V.** +31 24 648-8444, Fax: +31 24 645-1981, Email: sales@aladdin.nl
Russia	**Aladdin Software Security R.D. Ltd.** +7 095 923-0588, Fax: +7 095 928-6781, Email: aladdin@aladdin.msk.ru

1-800-223-4277

www.aks.com

ALADDIN

The Professional's Choice

Australia Conlab 03 98985685 ■ **China (Beijing)** Feitian 010 62567389 **(Hong Kong)** Hastings 02 5484629 **(Shenzhen)** Hastings 0755 2328741 ■ **Czech** Atlas 02 766085 ■ **Denmark** Berendsen 039 577316 ■ **Egypt** Zeineldein 02 3604632 ■ **Finland** ID-Systems 09 8703520 ■ **Greece** Unibrain 01 6756320 ■ **India** Solution 011 2148254 ■ **Italy** Partner Data 02 26147380 ■ **Jordan** ProgressSoft 06 5623820 ■ **Korea** Dae-A 02 8484481 ■ **Mexico** SiSoft 091 80055283 ■ **Poland** Systherm 061 8480273 ■ **Portugal** Futurmatica 01 4116269 ■ **Romania** Ro Interactive 092 353645 ■ **Singapore** ITR 065 5666788 ■ **South Africa** Adroit 011 8864704 ■ **Spain** PC Hardware 03 4493193 ■ **Sweden** Kordab 455 307 300 ■ **Switzerland** Opag 061 7169222 ■ **Taiwan** Teco 02 25559676 ■ **Turkey** Mikrobeta 0312 4670635

* 1998 BSA/SPA Study

GETTING STARTED

by Dave Mark and Dan Parks Sydow

Color Animation

How a Mac program generates smooth, flicker-free color animation

A couple of articles back we covered the bitmap and its use in offscreen drawing to create smooth black and white animation. Last month's article introduced Color QuickDraw and drawing in color. In this month's article we combine all these recently learned techniques to tackle smooth, fast, color animation.

Bitmaps and Offscreen Drawing Review

A *bitmap* is a representation of a monochrome (blacK-and-white) image. The map is composed of a grid of pixels, with each pixel considered either on or off. An image is defined by specifying the state of each pixel in the map. A single bit is used to keep track of whether a single pixel is on or off.

An *offscreen bitmap* is a bitmap that is drawn in memory alone — it doesn't have an onscreen representation. That is, a data structure holds the bit information that defines an image, but the contents of this data structure aren't translated to a window's graphics port. Animation is accomplished using a total of three offscreen bitmaps. One bitmap holds a background image, a second bitmap holds the foreground image, and the third bitmap is a *mixer*, or *master*, that is a combination of the other two bitmaps. The image that is the foreground bitmap and the image that is the background bitmap combine in the master bitmap offscreen (in memory), after which the image in the master (and only the master) bitmap appears onscreen. Repeating this process via a loop, with a slight shift of the position of the foreground image relative to the background image, is the basis for animation. Because each pass through the loop creates a master bitmap behind the scene in memory rather than in view of the user onscreen, flicker is kept to a minimum.

PixMaps and Offscreen Drawing

For monochrome animation, the BitMap data structure is used to define the state of the pixels that make up a bitmap image. Here's the BitMap data structure:

```
struct BitMap
{
  Ptr      baseAddr;
  Short    rowBytes;
  Rect     bounds;
};
```

A color image necessitates the use of a different data structure. The PixMap data structure includes the same three fields as the BitMap data structure — but it also holds more information as well. Among the extra information stored in the more complex PixMap is the number of bits used to define the color of each pixel. Here's a look at the PixMap data structure:

```
struct PixMap {
  Ptr          baseAddr;     /*pointer to pixels*/
  short        rowBytes;     /*offset to next line*/
  Rect         bounds;       /*encloses bitmap*/
  short        pmVersion;    /*pixMap vers number*/
  short        packType;     /*defines packing */
  long         packSize;     /*length of data*/
  Fixed        hRes;         /*horiz. Res. (ppi)*/
  Fixed        vRes;         /*vert. Res. (ppi)*/
  short        pixelType;    /*defines pixel type*/
  short        pixelSize;    /*bits in pixel*/
  short        cmpCount;     /*components in pixel*/
  short        cmpSize;      /*bits per component*/
```

Dan Parks Sydow is the author of over a dozen programming books, including "*Foundations of Mac Programming*" by IDG Books. Dan's lending a hand on this Getting Started article.

```
#if OLDPIXMAPSTRUCT
   long            planeBytes;   /*plane*/
   CTabHandle      pmTable;      /*color map*/
   long            pmReserved;
#else
   OSType          pixelFormat;  /*fourCharCode rep.*/
   CTabHandle      pmTable;      /*color map*/
   PixMapExtHandle pmExt;        /*pixMap handle ext.*/
#endif
};
```

With monochrome bitmap animation, all you had to do was fill out the relatively simple BitMap structure, create a GrafPort, then connect the two via a call to SetPortBits(). Once that's done, you are ready to copy the BitMap from port to port via a call to CopyBits().

A PixMap is more complex than a BitMap, so color pixel map animation requires extra effort. Fortunately, the Toolbox offers a high-level set of functions that simplify somewhat the creation of offscreen PixMaps. An offscreen PixMap is known as a *graphics world*, or GWorld. A GWorld is created in memory and typically drawn to a window via a call to CopyBits(), just as done with a BitMap.

The GWorld is a full-color, offscreen drawing environment. Just as you'd use an offscreen GrafPort and BitMap to prepare a black-and-white image for blitting to the screen (blitting comes from *BLock Transfer*, meaning copying a block of memory from one area to another, all at once), you'll use a GWorld to do the same for a color image.

PixMapper

Two issues ago, the *Getting Started* program BitMapper was developed to demonstrate offscreen animation using black-and-white bitmaps. **Figure 1** shows the window displayed by BitMapper. The floating hand is the foreground image and the framed gray pattern is the background image. As you move the mouse, the hand appears to float over the gray background, just like a cursor. What the user is seeing is the mixer bitmap — the copied version of the offscreen bitmap that represents the combining of the foreground hand bitmap with the background gray bitmap.

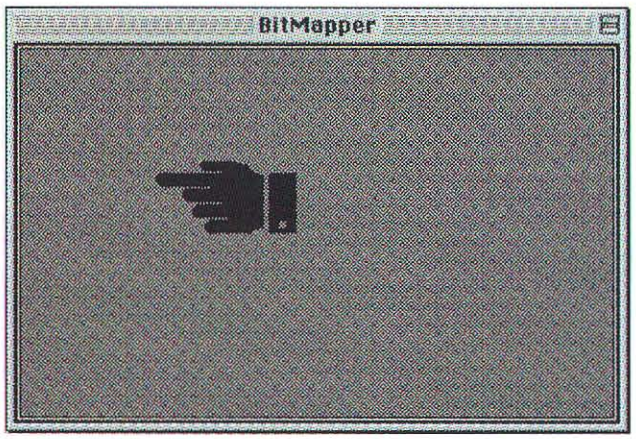


Figure 1. *The BitMapper window.*

This month's program is called PixMapper. Like BitMapper, PixMapper moves a foreground image over a background image. As we did for BitMapper, in PixMapper we create a PICT resource to serve as the foreground image. To demonstrate a different technique, though, in PixMapper we create the background image in our code. **Figure 2** shows the PICT resource (an image we copied from the Scrapbook) moving over the background (which is simply a checkerboard pattern drawn with few calls to QuickDraw routines).

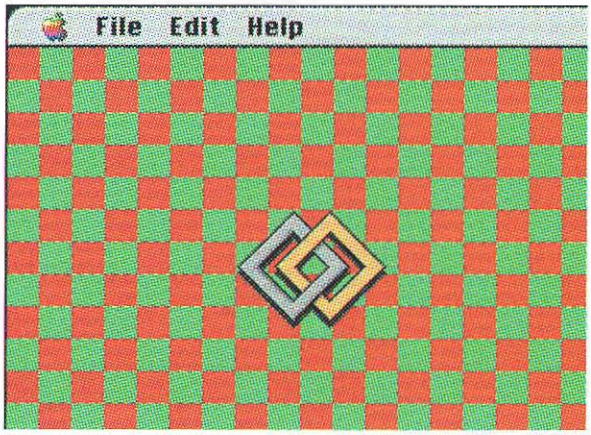


Figure 2. *PixMapper in action.*

As soon as you run PixMapper the menu bar, featuring the **Apple**, **File**, **Edit**, and **Help** menus, appears. With the exception of the **Quit** item in the **File** menu, there's nothing of significance in these menus. Next, a window appears, filling the entire main screen (the screen with the menu bar). PixMapper fills the window with a checkerboard pattern of red and green colored squares (just in time for Christmas, of course!). PixMapper then loads a PICT resource, and uses a series of offscreen GWorlds to animate the PICT across the colored background. The animation begins in the upper left corner and moves towards the lower right. Every time the PICT hits the edge of the window, the PICT bounces off and continues in the opposite direction.

The speed of the foreground image depends on the speed of your machine and the size of the PICT. The important thing to notice is that the PICT animates smoothly with absolutely no flicker. If there is any hesitation, it is most likely due to the system taking time to do some housekeeping chore.

Creating the PixMapper Resources

To get started, open your CodeWarrior development folder and create a folder named PixMapper. Start up ResEdit and create a new resource file named PixMapper.rsrc inside the PixMapper folder. **Figure 3** shows the five types of resources used by PixMapper. By now you should be experienced in creating and working with each of these resource types.

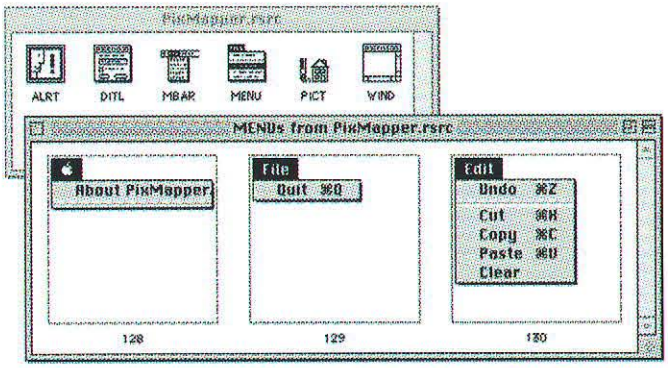


Figure 3. *The PixMapper resources.*

Figure 3 shows the three MENU resources the program needs. Only the **Quit** item in the **File** menu is of significance — the other items exist in preparation for turning PixMapper into a "real" program.

PixMapper uses the same ALRT and DITL resources that have been used in recent *Getting Started* examples. The one ALRT and one DITL are used to support the error-handling alert displayed by the program's DoError() routine (see the September 1998 Apple Events *Getting Started* column for more information on these resources and on the DoError() routine).

The one WIND resource will be used to display the color animation. The size of the WIND isn't at all important — we'll be resizing the window from within the source code. Since the window will be fixed on the screen, the type of WIND isn't too important either, though you'll want to choose a type that foregoes the drag bar (title bar) so the user doesn't get the impression that the window is movable.

You need a single PICT resource to serve as the foreground image. In the PixMapper source code we'll be referencing this resource by an ID of 128, so make sure to assign the PICT that value. For the foreground image you'll want something relatively small. Of course to witness the power of the PixMapper program you'll want to use a color image. You might also consider using a picture that is non-rectangular or that has a hole in it. In your paint program use the lasso tool to select only the pixels in the picture (and not the background). Though a rectangular picture will work just fine, a non-rectangular picture (like an X or an O shape) produces much more impressive results.

That's it for the PixMapper.rsrc file. Now quit ResEdit, making sure to first save your changes.

Creating the PixMapper Project

Launch CodeWarrior and create a new project based on the MacOS:C_C++:MacOS Toolbox:MacOS Toolbox Multi-Target stationary. You should have already created a project folder, so uncheck the Create Folder check box. Name the project PixMapper.mcp and designate the PixMapper folder as the project's destination.

Remove the SillyBalls.c and SillyBalls.rsrc placeholder files from the new project window, then add the PixMapper.rsrc file. The PixMapper project doesn't use of any of the standard ANSI libraries, so feel free to remove the ANSI Libraries folder.

Now choose **New** from the **File** menu to create a new, empty source code window. Save it with the name PixMapper.c and then choose **Add Window** from the **Project** menu to add the file to the project. The full source code listing for the PixMapper program appears next in the source code walk-through. You can type it into the PixMapper.c file as you read the walk-through, or you can take a shortcut and download the entire PixMapper project from MacTech's ftp site at <ftp://ftp.mactech.com/src/>.

Walking Through the Source Code

On to the code. As with other *Getting Started* projects, PixMapper starts off with some constant definitions.

```
/****************** constants ******************/

#define kMBARResID          128
#define kALRTResID          128
#define kWINDResID          128

#define kSleep              7
#define kMoveToFront        (WindowPtr)-1L

#define kSquareSize         16

#define kForegroundPICT     128
#define kIgnored            nil
#define kUseMaxDepth        0

#define mApple              128
#define iAbout              1

#define mFile               129
#define iQuit               1
```

There is a bunch of globals used by PixMapper. The familiar gDone starts life as false and is set to true when **Quit** is selected from the **File** menu. The program's only window is kept track of using the WindowPtr variable gMainWindow. Variables gXBump and gYBump specify the number of pixels the PICT moves each new animation cycle. One way to speed up the animation is to raise the bump values, though the code was written to work with single pixel movements.

```
/***************** global variables ****************/

Boolean         gDone;
WindowPtr       gMainWindow;
short           gXBump = 1, gYBump = 1;
```

We'll use three GWorlds. The graphics world gPictWorld holds the PICT image. Graphics world gSaveWorld holds the background of the window for later restoration. Finally, gSaveMixWorld is used to combine the PICT and the background. The three PixMapHandles are handles to the PixMaps tied to their respective GWorld.

```
CWorldPtr       gPictWorld;
CWorldPtr       gSaveWorld;
CWorldPtr       gSaveMixWorld;
PixMapHandle    gPixMapSave;
PixMapHandle    gPixMapSaveMix;
PixMapHandle    gPixMapPict;
```

The rectangle gPictWorldRect is the exact size of the PICT and is the bounding rectangle for gPictWorld. Rectangle gWorldRect is the bounding Rect for the mixing GWorld. The mixing GWorld is one pixel bigger in all directions than the PICT. This was done so

that when we save a region the size of gWorldRect from the PixMapper window, we'll have a one pixel border around the PICT. This way, when the PICT moves one pixel in any direction, we'll have the right pixels saved for later restoration. You'll see how this works later in the code. Rectangle gSavedFloaterRect contains the current position of the PICT in the PixMapper window.

```
Rect        gPictWorldRect;
Rect        gWorldRect;
Rect        gSavedFloaterRect;
```

Next come the program's function prototypes:

```
/****************** functions ******************/
void        ToolBoxInit( void );
void        MenuBarInit( void );
Boolean     HasGWorlds( void );
void        CreateWindow( void );
void        PaintWindow( void );
void        GWorldInit( void );
GWorldPtr   MakeGWorld( Rect *boundsPtr );
void        DrawFirstFloater( void );
void        MoveFloater( void );
void        CalcNewFloaterPosition( void );
void        EventLoop( void );
void        DoEvent( EventRecord *eventPtr );
void        HandleMouseDoWn( EventRecord *eventPtr );
void        HandleMenuChoice( long menuChoice );
void        HandleAppleChoice( short item );
void        HandleFileChoice( short item );
void        DoError( Str255 errorString );
```

As always, main() begins by initializing the Toolbox.

```
/********************* main *********************/

void main( void )
{
  ToolBoxInit();
```

Next, HasGWorlds() is called to see if GWorlds are available on this machine. We'll look at HasGWorlds() just ahead.

```
  if ( ! HasGWorlds() )
    DoError( "\pDeep GWorlds not supported" );
```

If so, the menu bar is set up, the main window is created, and the three graphics worlds are created. Both CreateWindow() and GWorldInit() are discussed later in this walk-through.

```
  MenuBarInit();
  CreateWindow();
  GWorldInit();
```

Next, the initial position of the PICT is plotted and the main animation loop is entered. The word *floater* refers to the foreground PICT, which appears to float over the background.

```
  DrawFirstFloater();

  EventLoop();
}
```

ToolBoxInit() and MenuBarInit() are the same as prior versions.

```
/****************** ToolBoxInit ******************/

void ToolBoxInit( void )
{
  InitGraf( &qd.thePort );
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs( NULL );
  InitCursor();
}

/****************** MenuBarInit ******************/

void MenuBarInit( void )
{
  Handle        menuBar;
  MenuHandle    menu;

  menuBar = GetNewMBar( kMBARResID );
  SetMenuBar( menuBar );

  menu = GetMenuHandle( mApple );
  AppendResMenu( menu, 'DRVR' );

  DrawMenuBar();
}
```

HasGWorlds() calls Gestalt() using the selector gestaltQuickdrawFeatures. If Gestalt() returns an error we'll display an appropriate error message.

```
/****************** HasGWorlds ******************/

Boolean  HasGWorlds( void )
{
  long   response;
  long   mask;
  OSErr  err;

  err = Gestalt( gestaltQuickdrawFeatures, &response );

  if ( err != noErr )
    DoError( "\pError calling Gestalt()" );
```

Next, we'll set up a comparison mask so we can look at the appropriate bit in response. Since gestaltHasDeepGWorlds has a value of 1, we'll want to look at bit number 1, which is the second bit from the right. We'll use the << operator to set bit number 1 in mask, leaving mask with a value of 2.

```
  mask = 1 << gestaltHasDeepGWorlds;
```

Finally, we'll use mask to see if bit number 1 is set in response. If so, deep GWorlds are available and we'll return true. Otherwise, we'll return false.

```
  if ( response & mask )
    return true;
  else
    return false;
}
```

CreateWindow() creates a new color window a little shorter than the main screen. The top of the window starts just below the menu bar. After setting up the window's size, a call to GetNewCWindow() creates a new CWindowRecord (as opposed to the WindowRecord that would result from a call to GetNewWindow()).

```
/****************** CreateWindow ******************/

void CreateWindow( void )
{
  Rect   wBound;
  long   wWidth, wHeight;
```

```
    wBound = qd.screenBits.bounds;
    wBound.top += GetMBarHeight();

    gMainWindow = GetNewCWindow( kWINDResID, nil,
                                 kMoveToFront);
```

The window is based on a WIND resource. Recall that we chose an arbitrary size for the window when creating this resource. Now it's time to match the window size to the size of the user's screen. The window boundary calculations are based of the wBound rectangle, which holds the display area of the graphics device (less the menu bar height). Finally, a call to PaintWindow() fills the window with colored rectangles.

```
    wWidth = wBound.right - wBound.left;
    wHeight = wBound.bottom - wBound.top;

    SizeWindow( gMainWindow, wWidth, wHeight, true );
    MoveWindow( gMainWindow, wBound.left, wBound.top, true );

    ShowWindow( gMainWindow );
    SetPort( gMainWindow );

    PaintWindow();
}
```

PaintWindow() starts by declaring several variables and calculating the number of columns and rows in the PixMapper window.

```
/****************** PaintWindow ******************/

void PaintWindow( void )
{
    RGBColor    redColor   = {65535, 0, 0};
    RGBColor    greenColor = {0, 40000, 15000};
    RGBColor    currentColor;
    Rect        r;
    short       row, col, numRows, numCols;

    SetPort( gMainWindow );

    r = gMainWindow->portRect;
```

Both numCols and numRows are based on kSquareSize. Each square on the window will be kSquareSize pixels on a side. If either numCols or numRows is not evenly divisible by kSquareSize, we'll add another row or column just so we don't leave any white space at the edge of the window.

```
    numCols = (r.right - r.left) / kSquareSize;
    if ( numCols != numCols/kSquareSize * kSquareSize )
      numCols++;

    numRows = (r.bottom - r.top) / kSquareSize;
    if ( numRows != numRows/kSquareSize * kSquareSize )
      numRows++;
```

Next, we'll step through all the squares, drawing each in the color appropriate to create a red and green checkerboard pattern. First, we set up the boundaries of a single square.

```
    for ( row = 0; row < numRows; row++ )
      for ( col = 0; col < numCols; col++ )
      {
        r.top = row * kSquareSize;
        r.bottom = r.top + kSquareSize;
        r.left = col * kSquareSize;
        r.right = r.left + kSquareSize;
```

Now, we determine whether the square should be red or green. Recall from your C background that the modulus operator (%) returns the remainder of an integral division. So, for example, if row is even and we divide by 2, the modulus result is 0 (no remainder). If row is odd when we divide by 2, the modulus is always 1. The following could be written a *little* more compact then it now appears, but the result would be even more confusing!

```
if ( ( row % 2 == 1 ) && ( col % 2 == 1 ) )
  currentColor = redColor;
else if ( ( row % 2 == 0 ) && ( col % 2 == 0 ))
  currentColor = redColor;
else
  currentColor = greenColor;
```

A call to RGBForeColor() is made to set up subsequent drawing in the appropriate color. Recall from last month that RGBForeColor() is the QuickDraw routine that accepts an RGB color as its parameter. A call to the QuickDraw function PaintRect() actually draws the colored rectangle.

```
    RGBForeColor( &currentColor );

    PaintRect( &r );
  }
```

When we're done, we set the foreground and background colors to their normal values. QuickDraw defines eight global constants to represent eight very basic colors (including black and white — refer to the QuickDraw.h universal header file for the rest). The QuickDraw routines ForeColor() and BackColor() are used with any of these eight constants to set the foreground color (the color used for drawing) and the background color (the color used to repaint a window's content area). Note that we could have used RGBForeColor() and RGBBackColor() here, provided we set up and specified black and white as RGBColor variables.

```
  ForeColor( blackColor );
  BackColor( whiteColor );
}
```

Now it's time to look at some code that involves graphics worlds. The three GWorlds are created in GWorldInit(), a routine that was called from main(). GWorldInit() starts by loading the PICT resource.

```
/********** GWorldInit **********/

void GWorldInit( void )
{
  PicHandle  pic;

  pic = GetPicture( kForegroundPICT );

  if ( pic == nil )
    DoError( "\pError loading PICT..." );
```

We'll grab the PICT's frame and normalize it (make its upper left corner (0,0)).

```
  gPictWorldRect = (**pic).picFrame;
  OffsetRect(   &gPictWorldRect, -gPictWorldRect.left,
                -gPictWorldRect.top );
```

gWorldRect is set to be 2 pixels taller and 2 pixels wider than the PICT. That leaves a one pixel border all the way around.

```
  gWorldRect = gPictWorldRect;
  gWorldRect.bottom += 2;
  gWorldRect.right += 2;
```

Next, we'll call our own MakeGWorld() routine to build one GWorld the size of the PICT and two the size of gWorldRect, storing the pointers in our three globals.

```
  gPictWorld = MakeGWorld( &gPictWorldRect );
  gSaveWorld = MakeGWorld( &gWorldRect );
  gSaveMixWorld = MakeGWorld( &gWorldRect );
```

When we create a GWorld, a PixMap is created for us. We'll call GetGWorldPixMap() to store the handle to each PixMap in its respective global.

```
  gPixMapPict = GetGWorldPixMap( gPictWorld );
  gPixMapSave = GetGWorldPixMap( gSaveWorld );
  gPixMapSaveMix = GetGWorldPixMap( gSaveMixWorld );
```

Next, we'll lock all three PixMaps in memory. Why? Just as you'd lock a handle before you singly dereferenced it to access its pointer, you lock your PixMap before you draw into it. Normally, you'd lock the pixels just before you draw, then unlock the pixels after the call to the drawing routine returns to prevent heap fragmentation. To keep things simple, we're just going to lock all three PixMaps for the duration of the program.

```
  if ( ! LockPixels( gPixMapPict ) )
    DoError( "\pLockPixels failed..." );

  if ( ! LockPixels( gPixMapSave ) )
    DoError( "\pLockPixels failed..." );

  if ( ! LockPixels( gPixMapSaveMix ) )
    DoError( "\pLockPixels failed..." );
```

Finally, we'll make the gPictWorld the current GWorld and draw the PICT in it. SetGWorld() makes the specified GWorld the current port, just as a call to SetPort() might make a window the current port.

```
  SetGWorld( gPictWorld, kIgnored );

  DrawPicture( pic, &gPictWorldRect );
}
```

MakeGWorld() calls NewGWorld() to create a new GWorld, returning a pointer to the new GWorld. The first parameter is the address of the GWorldPtr that will eventually point to the new GWorld. The second parameter specifies the pixel depth of the new GWorld. By passing in a value of 0, we're asking NewGWorld() to use the deepest device that intersects boundsPtr, the third parameter. The fourth and fifth parameters specify a color table and a GDevice, in case you want to roll your own. We'll pass nil in for each, asking NewGWorld() to take care of these parameters for us. The final parameter lets us set special GWorld flags. We'll pass in 0, ignoring the flags. You can read about these flags in the description of NewGWorld() in the *Offscreen Graphics Worlds* chapter *of Inside Macintosh: Imaging With QuickDraw.*

```
/********** MakeGWorld **********/

GWorldPtr MakeGWorld( Rect *boundsPtr )
{
  QDErr      err;
  GWorldPtr newGWorld;
```

```
    err = NewGWorld( &newGWorld, kUseMaxDepth,
          boundsPtr, kIgnored, kIgnored, noNewDevice );

    if ( err != noErr )
      DoError( "\pMy call to NewGWorld died! Bye..." );

    return( newGWorld );
}
```

Here, in DrawFirstFloater(), comes the really important stuff. Just as it did in BitMapper, CopyBits() is used to copy a block of pixels from one offscreen to another. Though CopyBits() expects to work with pointers to BitMaps, it can handle either BitMaps or PixMaps. A bit of typecasting is necessary, however, if only to placate the compiler's typecasting mechanism.

```
/********** DrawFirstFloater **********/

void DrawFirstFloater( void )
{
```

Each call to CopyBits() copies from the first parameter to the second, using the Rects in the third and fourth parameters. The srcCopy mode tells CopyBits() to replace all the destination bits with the appropriate source bits. The transparent mode, on the other hand, tells the compiler not to copy the white pixels. This comes in handy when we copy a non-rectangular or non-solid image from one GWorld to another. The last parameter to CopyBits() specifies an optional mask parameter which we won't use. Passing nil tells CopyBits() to ignore this parameter.

The first call to CopyBits() copies the background of the PixMapper window into the gPixMapSave PixMap. We're saving away the pixels we're about to obliterate with the PICT, with an extra one pixel border we'll need when the floater moves in one direction or the other.

```
    CopyBits(   &(gMainWindow->portBits),
                (BitMap *)(*gPixMapSave),
                &gWorldRect, &gWorldRect, srcCopy, nil );
```

Next, we'll set up a Rect the size of the PICT that is 1 pixel down and 1 pixel to the right of the upper left corner of the window. This is where we'll plot the PICT.

```
    gSavedFloaterRect = gPictWorldRect;
    OffsetRect( &gSavedFloaterRect, 1, 1 );
```

This call to CopyBits() draws the PICT in the PixMapper window.

```
    CopyBits(   (BitMap *)(*gPixMapPict),
                &(gMainWindow->portBits),
                &gPictWorldRect, &gSavedFloaterRect,
                transparent, nil );
}
```

MoveFloater() is responsible for moving the foreground image. Because the image is moved only slightly, MoveFloater() gets called repeatedly from the event loop. MoveFloater() starts off by calling CalcNewFloaterPosition() to update the values of gXBump and gYBump, in case the floater is hitting the edge of the window.

```
/********** MoveFloater **********/
```

```
void MoveFloater( void )
{
  Rect   r;
  RgnHandle newRgn, savedRgn, oldClip;

  CalcNewFloaterPosition();
```

This call to CopyBits() copies the saved pixels to the mixer GWorld.

```
  CopyBits(   (BitMap *)(*gPixMapSave),
              (BitMap *)(*gPixMapSaveMix),
              &gWorldRect, &gWorldRect, srcCopy, nil );
```

Next, we position a Rect the size of the PICT in the mix GWorld using gXBump and gYBump. This Rect is the new position of the floater in the mix GWorld. We'll then call CopyBits() to copy the floater on top of the saved pixels in the mixing GWorld. Remember, we used transparent mode so we'd only draw the non-background pixels. You might want to handle this differently if you need to draw white pixels.

```
  r = gPictWorldRect;
  OffsetRect( &r, gXBump + 1, gYBump + 1 );

  CopyBits(   (BitMap *)(*gPixMapPict),
              (BitMap *)(*gPixMapSaveMix),
              &gPictWorldRect, &r, transparent, nil );
```

Next, we construct a Rect at the floater's last position in the PixMapper window, then make it one pixel bigger in all directions (the size of the mixing GWorld). We're going to use this Rect to copy the contents of the mixing GWorld into the window.

```
  r = gSavedFloaterRect;
  InsetRect( &r, -1, -1 );

  CopyBits(   (BitMap *)(*gPixMapSaveMix),
              &(gMainWindow->portBits),
              &gWorldRect, &r, srcCopy, nil );
```

Now we update the saved floater position stored in gSavedFloaterRect to reflect the new position.

```
  OffsetRect( &gSavedFloaterRect, gXBump, gYBump );
```

Following that, we create our one pixel bigger Rect again, this time at the floater's new position. We then copy the floater, with a one pixel border, into the mixing GWorld.

```
  r = gSavedFloaterRect;
  InsetRect( &r, -1, -1 );

  CopyBits(   &(gMainWindow->portBits),
              (BitMap *)(*gPixMapSaveMix),
              &r, &gWorldRect, srcCopy, nil );
```

Next, copy the saved pixels into the appropriate position in the mixing GWorld. The idea here is that we are reconstructing the pixels that should be behind the floater.

```
  r = gWorldRect;
  OffsetRect( &r, -gXBump, -gYBump );

  CopyBits(   (BitMap *)(*gPixMapSave),
              (BitMap *)(*gPixMapSaveMix),
              &gWorldRect, &r, srcCopy, nil );
```

Finally, we copy the reconstructed "behind the floater" pixels from the mix GWorld into the save GWorld. We are now ready to move the floater all over again.

```
  CopyBits(   (BitMap *)(*gPixMapSaveMix),
              (BitMap *)(*gPixMapSave),
              &gWorldRect, &gWorldRect, srcCopy, nil );
}
```

This routine figures out if bumping the floater will move it off the edge of the window in any direction. If so, the direction of floater movement is changed, so the floater moves away from that edge, rather than towards it.

```
/************** CalcNewFloaterPosition **************/

void CalcNewFloaterPosition( void )
{
  Rect r;

  r = gSavedFloaterRect;

  OffsetRect( &r, gXBump, gYBump );

  if ( (r.left < gMainWindow->portRect.left) ||
       ( r.right > gMainWindow->portRect.right ) )
    gXBump *= -1;

  if ( (r.top < gMainWindow->portRect.top) ||
       ( r.bottom > gMainWindow->portRect.bottom ) )
    gYBump *= -1;
}
```

At this point we're home free. The remaining code, with a single exception, is all "copy and paste" code from other *Getting Started* examples. The exception appears in EventLoop(). It's here that we repeatedly call MoveFloater() to keep the animation running. Only when the user quits does the animation loop end.

```
/********************* EventLoop *********************/

void EventLoop( void )
{
  EventRecord   event;

  gDone = false;
  while ( gDone == false )
  {
    if ( WaitNextEvent( everyEvent, &event, kSleep, nil ) )
      DoEvent( &event );

    MoveFloater();
  }
}

/********************** DoEvent **********************/

void DoEvent( EventRecord *eventPtr )
{
  char theChar;

  switch ( eventPtr->what )
  {
    case mouseDown:
      HandleMouseDown( eventPtr );
      break;
    case keyDown:
    case autoKey:
      theChar = eventPtr->message & charCodeMask;
      if ( (eventPtr->modifiers & cmdKey) != 0 )
        HandleMenuChoice( MenuKey( theChar ) );
      break;
    case updateEvt:
      BeginUpdate( (WindowPtr)(eventPtr->message) );
      EndUpdate( (WindowPtr)(eventPtr->message) );
      break;
```

How much do you pay for network consultation?

How does FREE sound?

Farallon®

Get Your **FREE** Personal Network Consultation with Dr. Farallon

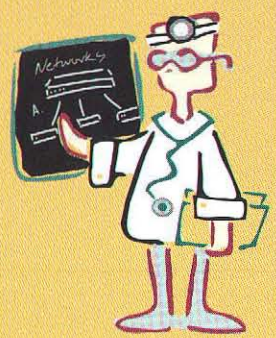
At absolutely no cost Dr. Farallon's network experts can help you...

- ✔ Implement switching to optimize your current network
- ✔ Upgrade your Macs, PCs, servers & printers to Ethernet or Fast Ethernet
- ✔ Integrate 10Base-T & 100Base-T with 10/100 hubs & switches
- ✔ Budget for network improvements & plan for network growth

Call: *1-800-613-4954*
Visit: *www.farallon.com/dr.farallon/mactech.html*

"For free, a Dr. Farallon Network Specialist analyzed our existing system, identified a range of options, and helped us plan and install a network with the speed and capacity we need at a price we could afford."

Irene Ogus, CEO, Media Corps



```
    }
}

/****************** HandleMouseDown ***************/

void HandleMouseDown( EventRecord *eventPtr )
{
  WindowPtr   window;
  short       thePart;
  long        menuChoice;

  thePart = FindWindow( eventPtr->where, &window );

  switch ( thePart )
  {
    case inMenuBar:
      menuChoice = MenuSelect( eventPtr->where );
      HandleMenuChoice( menuChoice );
      break;
    case inSysWindow :
      SystemClick( eventPtr, window );
      break;
  }
}

/****************** HandleMenuChoice ***************/

void HandleMenuChoice( long menuChoice )
{
  short  menu;
  short  item;

  if ( menuChoice != 0 )
  {
    menu = HiWord( menuChoice );
    item = LoWord( menuChoice );

    switch ( menu )
    {
      case mApple:
        HandleAppleChoice( item );
        break;
      case mFile:
        HandleFileChoice( item );
        break;
    }
    HiliteMenu( 0 );
  }
}

/****************** HandleAppleChoice ***************/

void HandleAppleChoice( short item )
{
  MenuHandle  appleMenu;
  Str255      accName;
  short       accNumber;

  switch ( item )
  {
    case iAbout:
      SysBeep( 10 );
      break;
    default:
      appleMenu = GetMenuHandle( mApple );
      GetMenuItemText( appleMenu, item, accName );
      accNumber = OpenDeskAcc( accName );
      break;
  }
}

/****************** HandleFileChoice ***************/

void HandleFileChoice( short item )
{
  switch ( item )
  {
    case iQuit:
      gDone = true;
      break;
  }
}

/********************** DoError ********************/

void DoError( Str255 errorString )
{
  ParamText( errorString, "\p", "\p", "\p" );

  StopAlert( kALRTResID, nil );

  ExitToShell();
}
```

Running PixMapper

Run PixMapper by selecting **Run** from the **Project** menu. Once your code compiles, a window appears with a checkerboard pattern drawn in it. The window will be the exact size of your screen. The animation begins automatically. As the foreground image moves, notice that the background shows through the image openings. Choose **Quit** from the **File** menu to end the animation and the program.

Till Next Month...

PixMap data structures and graphics worlds can be tricky to work with, but the PixMapper code should give you a good base for creating your own animated effects. Don't worry too much about the specifics of the PixMapper algorithm. The important things to understand are how to construct a GWorld, how to use CopyBits() to copy PixMaps between GWorlds and windows, and the basics of working with color. You can get more information on the PixMap data structure in the *Color QuickDraw* chapter of *Inside Macintosh: Imaging With QuickDraw*. The *QuickDraw Drawing* chapter of that same volume discusses the all-important CopyBits() routine. For more on graphics worlds, read the *Offscreen Graphics Worlds* chapter of *Imaging With QuickDraw*.

If you want to turn PixMapper into a more useful program, provide the user with some control of the animation. PixMapper's MoveFloater() routine is responsible for creating one "frame" in an animated sequence, and is called repeatedly from EventLoop(). Be aware that while a routine like MoveFloater() needs to be called from within a loop to achieve an animated effect, that loop doesn't have to be the event loop. Try adding an **Animation** menu and then moving the call to a routine that handles a selection from that menu.

If PixMapper seems somewhat slow to you, there's a reason for that. CopyBits() is a general purpose routine designed to handle all color environments and both BitMaps and PixMaps. There are also many ways you can improve the performance of the basic PixMapper algorithm, though that tends to make the code even more confusing. One simple way to speed things up a bit is to increase the value of both the gXBump and gYBump global variables. To maintain smooth animation, you'll also have to alter a little bit of the code. Hint: It's no coincidence that as written, gXBump and gYBump both have a value of 1, and the global variable gWorldRect is set up to be one pixel larger in each direction then the foreground image.

These suggestions should provide you with plenty to do until next month's column. See you then...

All the information you need in the palm of your hand.

Now with the THINK Reference Compiler!

Almost 1800 articles from over 150 issues of *MacTech Magazine* (1984 through May of 1998). Improved hypertext, improved indicies, and a new THINK Reference Viewer for lightning quick access! Full version of THINK Reference, the best threads from Macintosh programmers, plus thousands of notes, tips, snippets, gotchas, and much, much more!

Available from

Developer DEPOT®

<http://www.devdepot.com>

PO Box 5200 • Westlake Village, CA • 91359-5200
Voice: 800/MACDEV-1 (800/622-3381) • Outside US/Canada: 805/494-9797
Fax: 805/494-9798 • E-mail: orders@devdepot.com

by John C. Daub, Austin, Texas, USA

What Is PowerPlant?

John would like to thank Dave Mark for encourging him to write this article, Carl, Richard, Kenny, and the PowerPlant user community for input and ideas, and also thank Greg and Vicki for kicking him when he needed it. He especially want's to thank his wife, Michele for believing in him, and his son, Wade, for making him get off the computer once in a while to watch *Toy Story* (again :-).

An introduction to and overview of the Metrowerks PowerPlant application framework

Introduction

Despite what many have said, I just don't see it. The Mac programming community doesn't seem to be dying (like many naysayers suggested last year). People are still flocking to the Mac (especially after the iMac introduction), and I still see newcomers every day in the comp.sys.mac.oop.powerplant USENET newsgroup — all with a burning desire to write the next "KillerApp." Many of the next generation programmers want to use Metrowerks PowerPlant to accomplish their goal, but quickly run into the same problem. PowerPlant is *enormous*, and can be intimidating — where and how do you start? I have heard this complaint from many aspiring PowerPlant users, and I can't blame them.

To try and help programmers become more familiar with PowerPlant, I decided to write a series of articles that will introduce various parts of the PowerPlant architecture to MacTech readers. These topics should be interesting to both new and experienced programmers. I know few people that understand all of PowerPlant (or even a large chunk), so I think there will be something for everyone.

This article will start with the very basics. We'll introduce frameworks in general, and then PowerPlant in specific. We'll talk about the advantages of using PowerPlant, and then describe the overall design of PowerPlant. We'll look into how event's are handled in PowerPlant, and how information is communicated between different parts of PowerPlant. You may see some simliarities to the first few chapters of the *PowerPlant Book*, Metrowerks' official documentation for PowerPlant — a book I highly recommend. In later articles, we'll get more "hands-on" and demostrate some of the real power of PowerPlant. But first, let's start with the basics.

Background

Definition

What is PowerPlant? The easy answer is: PowerPlant is Metrowerks Corporation's object-oriented application framework for Mac OS software development. It is written in the C++ programming language, and takes full advantage of the features C++ offers (such as multiple inheritance, polymorphism, templates, and namespaces). PowerPlant provides programmers with a collection of solid, field-tested, code that can serve as a mature foundation upon which to build your software product — letting you build more reliable applications, in less time.

Not only does PowerPlant help you lay a rock-solid foundation for your application, but PowerPlant also provides a wealth of tools beyond what you would expect from a simple application framework. For example, there are TCP/IP networking classes; debugging classes to help write better code;

John C. Daub is currently one of Metrowerks' PowerPlant Engineers. Since he joined Metrowerks in 1996 he has been involved with PowerPlant in some capacity or other, from technical support to quality assurance to authoring. If you have questions about PowerPlant, or if you have any good lawn care tips (especially if you're familiar with Texas soils) you can contact John at hsoi@metrowerks.com.

classes that simplify threading and extend the capabilities of the Thread Manager; and classes for all the major Internet protocol implementations (POP3, SMTP, FTP, HTTP, Finger, NTP, IRC). PowerPlant offers flexible implementations of and support for Mac OS technologies like Drag and Drop, the Appearance Manager, and the Contextual Menu Manager. PowerPlant includes cool things like Attachments, that world-class Metrowerks Technical Support, and will comply with and support Carbon as Carbon develops and evolves. That's just the short list.

Pretty neat, huh? That list makes it seem like PowerPlant can do a lot. But there are a lot of buzzwords and technical jargon in it that might not be familiar to you. Let's break this definition into smaller pieces and examine each piece in turn.

PowerPlant Prerequisites

PowerPlant assumes two fundamental skills: you need to be able to use the C++ programming language and be able to use the Mac OS Toolbox.

PowerPlant is written in the ANSI/ISO C++ programming language. Earlier I mentioned some wonderful C++ language features: **namespace**, **template**, **polymorphism**, and **multiple inheritance**. You don't need to know anything about them. They're certainly powerful features and if you're curious you might consider buying a copy of the recently drafted ANSI/ISO C++ standard for those tough questions that will occasionally arise. You do need to be able to write basic C++, and you'll need to know how to use inheritence explore PowerPlant. But that's about it. Beginners are certainly welcome here, and you will assuredly sharpen your C++ skills as you work with the language.

PowerPlant simplifies and hides some of the OS-level details from you. For instance, the Networking Classes unify the two TCP/IP mechanisms (MacTCP and Open Transport) into a single API (Application Programmer's Interface) for TCP/IP communication. Because you only have to talk to the single interface (PowerPlants interface), you can write less code while still offering your application to many different kinds of users. However, you can't always work from this higher level of abstraction. At times you will have to get some dirt under your fingernails and dig into the Toolbox directly. This doesn't mean you have to be an expert on the Toolbox, just make sure you understand the basics of how the Mac OS works and how to work with the Toolbox when you need to. There is a lot of documentation and sample code out there to help you, as well as a very open, welcoming, and helpful community of other Mac OS developers on forums such as the comp.sys.mac.programmer.* newsgroup hierarchy.

If you need some help becoming familiar with C++ or understanding how the Mac OS Toolbox works, there are lots of good books at your local bookstore, or perhaps your local library. If you can lay a good foundation for yourself, it will be a lot easier further down the road to work with the framework.

So What is a Framework?

PowerPlant is a framework... but what is a framework? Before we can answer that question, we need to provide some background and discuss code reuse and the various types of reusable code.

It takes time to run a business, time is money, and a business needs and wants to make money. If something could be done to make more and expend less, typically a business would find this "thing" a positive means. Many typical software development tasks have already been done by some party somewher. Fortunately many authors of these codes are willing to make their solutions available for others to use; some as commercial ventures, others as free and open source. These codes are developed over the course of many months and/or years, which enables them to be field tested, reliable, focused, and most importantly, (re)usable. If it took these parties months or years to develop these codes into mature products, how long would it take you to reinvent the wheel? And does that provide the best cost-benefit ratio for your company? Depending upon your situation perhaps it might — each and every situation must be evaluated on its own merits under its unique circumstances. But chances are better that the reuse of code will provide the bigger payoff. You might want to read § 23.5.1 of Bjarne Stroustrup's *The C++ Programming Language* (3rd edition). As the creator of C++, Stroustrup raises some interesting issues about code reuse. Besides, you need to beat your competition to market, and reusable code could give you that advantage. Reusable code is no panacea (what is?), but it can be a Good Thing.

There are various types of reusable code. There are the random examples and snippets that can be found in places like collection CD's (Apprentice) and websites (Apple, alt.sources.mac). Although these examples and snippets tend to be unorganized and random (at least compared to other types of reusable code), nonetheless they can be quite useful at solving specific problems that can arise during the course of development.

A **procedural code library** is a more ordered collection of reusable code snippets, routines, and/or utilities. The ANSI C Standard Library is an example of this type of reusable code. It contains routines for string manipulations (strcpy, strlen), file i/o (fopen, fwrite), mathematics (sin, cos, log), and sundry utilities (rand, qsort), amongst other things. Libraries, such as the C Standard Library, are very useful indeed. However, as procedural code libraries tend to try to provide a higher degree of reuse by addressing the lowest common denominator, they tend to be more generic and of a utility nature than some other types of reusable code, such as class libraries.

A **class library** provides a similar sort of utility as a procedural code library, but it also attempts to provide greater flexibility than a procedural library. A class library achieves this by virtue of being written in an object-oriented language, such as C++ or Java. The ANSI C++ Standard Library is an example of this type of reusable code. It provides strings (string), i/o (iostream, fstream), numerics (valarray, complex), and sundry utilities (typeinfo, vector), just like the Standard C Library does (which, by the way, is actually a subset within the C++ Standard Library). But these portions of the C++ Standard Library provide greater flexibility to you through use of C++ templates. "Templates provide a simple way to represent a wide range of general concepts and simple ways to combine them. The resulting classes and functions can match hand-written more-specialized code in run-time and space efficiency. Templates provide direct support for generic programming." (Stroustrup 1997). So you can see how a class library can have benefits over a procedural code library, but at least compared to a framework, a class library remains fairly generic.

A **framework** is a structured collection of routines (functions, methods, procedures), typically written in an object-oriented language, designed for reuse to aid in the solving of a specific software development problem. Of the various types of reusable code, it tends to be more specific in nature, aimed towards the solving of a particular problem set. In PowerPlant's case, this would be aiding in the development of software for the Mac OS. In terms of reusable code, the PowerPlant framework might not be as extensible as a more generic class library (e.g. PowerPlant does not target Unix very well), but because PowerPlant has a more directed focus (i.e. software for Mac OS), the job that it does it does well.

To clarify a bit more, PowerPlant is actually an **application framework**. Frameworks aim to solve specific problems, such as database solutions, multimedia packages, games, and so forth. An application framework is designed to aid in the authoring of applications. The framework provides some core structure (LEventDispatcher, LCommander, LBroadcaster); means for tapping into, and extending or even limiting that structure (LAttachment, LPeriodical); interface tools (LPane, LMenu), and useful utilities (UMemoryMgr, StDialogHandler). What a framework lacks is the unique content that makes an application your application — no need to spend time, money, nor brain power on anything but the specifics of your Killer App.

Of course with all good there is some bad. PowerPlant, as specific as the above might make it sound, is still somewhat generic. It attempts to address the needs of the majority of its users — not everyone will find exactly what they need immediately available in PowerPlant (sometimes called the "80/20 Rule", or "you cannot please everyone"). To work within these constraints, PowerPlant attempts to provide as much flexibility as possible to the user so that you can construct your own custom solution. We will discuss some of the designs in the next section.

As an application framework, PowerPlant offers a great deal. It provides you with a solid foundation. It provides you with basic as well as advanced tools to make your code more efficient and more robust, while saving you development time and resources. Just how it can provide these savings is the subject of the next section.

PowerPlant Design Patterns

At first, PowerPlant seems like a huge folder with hundreds of files and thousands upon thousands of lines of code — how can anyone make sense of this? Where do you start? Thankfully, PowerPlant is very well organized and designed; all you need is a roadmap to help you find your way through the various design patterns that PowerPlant employs. As you travel down the road, these basic patterns should begin to make sense to you. As you begin to understand the basic patterns, look for how the patterns work together to form the framework and provide to you the benefits mentioned above. The concepts we will discuss here are: event handling, targets and command chain, visual interface, interobject communication, persistence, and utilities.

Event Handling

Mac OS applications utilize an **event-driven programming model**. This model puts the application user in control, and the actions of the user drive the application. There are a known set of possible events, such as mouse clicks and keystrokes. When one of these events occur, the operating system (Event Manager) notifies your application of the event and provides important information about the event. For example, when you click the mouse, the Event Manager will provide your application with the coordinates of where the mouse was when the click occurred. When your application receives this information, it must respond appropriately (show a menu, display the typed character, etc.).

Because event handling and dispatch is an essential part of any Mac OS application, it is a perfect candidate for a framework. PowerPlant provides this core event handling through the LEventDispatcher class. LEventDispatcher has methods to dispatch and handle each event type appropriately. The class also provides for typical event-related behaviors such as adjusting the look of the cursor and performing menu updates.

LEventDispatcher only knows how to start the event down a specific event path. If the application user presses a key on their keyboard, LEventDispatcher starts the ball rolling to process that keystroke, but it does not process the keystroke itself. Suppose you had numerous text fields in your application's window? How does LEventDispatcher know which text field to direct the keystroke towards? Enter the concept of targets and command chains.

Targets and Command Chain

When you select a command from a menu or press a key on the keyboard, this event is dispatched to the target for handling. A **target** is the object to which most (if not all) directed events are dispatched . There is at most one target at any time. The target also helps to determine the state of the applications' menus. For example, if an editable text field (LEditField) was the current target and there was text on the clipboard that could be pasted into the field, the **Paste** menu item would be enabled. If the LEditField contained a selection, the **Cut**, **Copy**, and **Clear** commands would be enabled as well. If the target changed to an LEditField that could not be edited, the **Edit** menu (and its items) would be disabled to reflect the field's read-only status. As commands are target-oriented, it is only logical that menu state is influenced by the target.

If the target is unable to process the given command, the command is passed to the next object in the command chain to see if it can process the command. A **command chain** is a list, or more typically a tree, of commanders. A **commander** is a object that can respond to a command. Within the chain of command, a commander can have **subcommanders** but has only one **supercommander** (except for the top-most commander, which has no supercommander). Depending where a commander falls in the command chain, it could be both a subcommander (to another supercommander) and a supercommander (to other subcommanders).

All of this command work is handled by PowerPlant's LCommander class. It handles command chain maintenance (adding, modifying, removing) and target management (switching, being, not being). An object that wishes to handle commands mixes in the LCommander class, and typically will override the FindCommandStatus() and ObeyCommand() methods. FindCommandStatus() is utilized by LEventDispatcher to determine the menu state, and ObeyCommand() performs the actual command handling.

As mentioned before, if the object cannot handle a command, the object passes the command to the next commander in the command chain. In PowerPlant the target is usually at the end of one of the command chain branches. If the target cannot handle the command, it passes the command to its supercommander to see if it could handle the command. If this object cannot handle the command, the command is passed to the object's supercommander. This is repeated until some object handles the command, or we have reached the end/top of the chain.

The bottom-up approach to the command chain has some advantages over the top-down approach. First, this provides objects the ability to stand alone. The object only has to specify

and handle commands that are relevant to itself, and then can pass all others to its supercommander (whatever that might be). There is no need to repatch the commander chain when you wish to add a new object into the chain, plus this aids in furthering good object-oriented design. Second, since the command chain is a tree, it is much easier to walk up the chain than down. Each commander has at most one supercommander, so there is only one path to the top.

Commanders are a very important part of PowerPlant. They are the main avenue by which the users' actions are dispatched and executed. LCommander provides a simple, yet powerful, means for the processing and management of commands and commanders. Since LCommander is provided as a mix-in class, any object that wants or needs to respond to user commands can do so. This use of multiple inheritance in PowerPlant enables those visual elements, like LEditField, to respond to user actions (furthering that event-driven programming model), and those elements that do not need to directly respond to user actions, like LCaption, remain as simple and streamlined as possible.

Visual Interface

One of the main goals of an application framework is to provide tools for creating and managing the application's interface. There are a multitude of tasks involved here including: creation of interface objects, establishing the proper drawing environment, actual rendering of the element on screen, managing coordinate schemes, and managing an object's relation to other objects in terms of location and appearance. Perhaps the most prominent feature of PowerPlant is the tools that it provides for management of the visual interface. These tools include Constructor, LPane and friends, and a host of utilities.

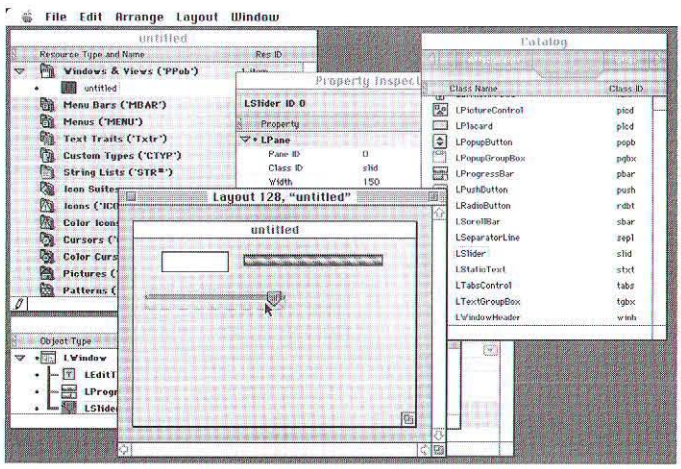


Figure 1. *Metrowerks Constructor.*

Constructor is the visual interface design and layout tool for PowerPlant (**Figure 1**). With Constructor you are able to design your screens in a WYSIWYG fashion. You can establish your windows and dialogs, their content, and the properties and behaviors of all of these elements. Constructor saves this layout information in a 'PPob' resource, which is a complex data structure that describes your screens. This data is then read and interpreted by PowerPlant to reanimate your screens at runtime (see UReanimator). And although Constructor is not a general-purpose resource editor like ResEdit or Resorcerer, it does allow you to edit PowerPlant resources ('Txtr', 'Mcmd'), some additional Mac OS resource types ('MENU', 'MBAR', 'STR#'), and bitmap resources (cursors, icons, PICTs, and patterns). We will take a more in-depth look at Constructor in a future article.

The basic class for all visual objects in PowerPlant is LPane. By itself, an LPane does not do much, but it does provide the basic form and function for all visual objects. In addition to the responsibilities mentioned at the beginning of this section, LPane also provides mechanisms for handling mouse clicks, obtaining the value and/or descriptor for an object (if such an attribute is relevant), coordination between the PowerPlant and Mac OS coordinate schemes, mouse tracking and cursor adjustment, and printing.

One behavior LPane does not have is the ability to manage subpanes in the visual hierarchy. Some interface objects do not need nor desire this ability, so why bloat those objects with additional and unnecessary code? For those objects that do need or want subpanes, there is LView. LView inherits from LPane, and essentially is a pane that can contain other panes. LView takes on most of the work involved in managing a pane's geographical relationship to other panes, and also provides support for scrolling a pane. Using LPane or LView as a base, you can create any interface object you might need; in fact, PowerPlant offers many common widgets: windows (LWindow), dialog boxes (LDialogBox), checkboxes (LCheckBox), radio buttons (LRadioButton), push buttons (LPushButton), popup menus (LPopupButton), scrollbars and scroller views (LScroller), text fields (LEditText, LTextEditView), sundry Appearance Manager controls (LProgressBar, LBevelButton) and so forth. And as you can see, PowerPlant's naming conventions try to be straightforward and logical. Readability of code is an important design principle in PowerPlant.

There is another design principle that permeates all of PowerPlant that is worth discussing here. This principle is **factoring**, which means that larger, more complex behaviors should be broken down into smaller, component parts. We have already discussed some of PowerPlant's factoring, such as the use of mix-in classes and multiple inheritance (LCommander, LEventDispatcher), and the relationship of LPane and LView. Hopefully these prior discussions have adequately demonstrated the benefits of a factored design.

An additional factored behavior used by PowerPlant is separating actions into a setup part and the actual action part. The setup part establishes any necessary states before the action executes. This could include state testing and modification, ensuring certain elements exist, and restoring states after the action has completed. The action part performs the actual activity. If setup proceeds with no problems, the setup then initiates the action. To perform the entire action, you call the setup method; usually you will not call the action method directly as performing the action without proper setup could lead to problems. In PowerPlant, the setup is performed by a function

with the same name as the function (e.g. Draw(), Click(), Show()). LPane::Draw() is shown in Listing 1.

Listing 1: LPane's Draw Function

LPane::Draw

```
void
LPane::Draw( RgnHandle inSuperDrawRgnH )
{
   Rect   frame;
   if ( (mVisible == triState_On) &&
      CalcPortFrameRect(frame) &&
      ((inSuperDrawRgnH == nil) ||
      ::RectInRgn(&frame, inSuperDrawRgnH)) &&
      FocusDraw() ) {

      PortToLocalPoint(topLeft(frame));
      PortToLocalPoint(botRight(frame));

      if (ExecuteAttachments(msg_DrawOrPrint, &frame)) {
         DrawSelf();
      }
   }
}
```

Draw() performs the generic setup actions that must be performed before any object draws itself. It ensures the object is visible, is within QuickDraw space, intersects the inSuperDrawRgnH, and can be focused. If all of these criteria are met, then and only then will it proceed to actually draw the object. The action, drawing the object in this case, is performed in a function named the same as its associated setup function with the word "Self" appended (e.g. DrawSelf(), ClickSelf(), ShowSelf()). Since LPane's implementation of DrawSelf() is empty, here is LCaption::DrawSelf(). LCaption is a subclass of LPane that draws a string of text.

Listing 2: LPane's Draw Function

LCaption::DrawSelf

```
void
LCaption::DrawSelf()
{
   Rect frame;
   CalcLocalFrameRect(frame);

   SInt16 just = UTextTraits::SetPortTextTraits(mTxtrID);

   RGBColor textColor;
   ::GetForeColor(&textColor);

   ApplyForeAndBackColors();
   ::RGBForeColor(&textColor);

   UTextDrawing::DrawWithJustification(
            (Ptr)&mText[1], mText[0], frame, just, true);
}
```

Due to the factored "Self" design, LCaption::DrawSelf() (listing oes not need to worry about general drawing setups — this is handled by LPane::Draw(), which LCaption inherits. LCaption only needs to trouble itself to establish its specific drawing needs (setting the text traits and fore/back colors) and then perform the core action of drawing the text string. To actually draw the string, just call the Draw() method:

```
// obtain a pointer to the LCaption object

LCaption *theCaption = dynamic_cast<LCaption*>
                        (theWindow->FindPaneByID(kCaptionID));

// draw the caption

theCaption->Draw(nil);
```

Most subclasses of LPane only need to override the DrawSelf() method to implement their specific drawing needs, leaving the housekeeping details to the framework. The same applies for other "Self" situations (override ClickSelf() and call Click() in response to a mouse click, override FinishCreateSelf() and call FinishCreate() to finish creating an PowerPlant visual object/element).

The final visual interface tool that we will address are interface utilities. We will discuss these utilities later in the article.

Interobject Communication

The commander mechanism is a specialized communication channel: it only communicates vertically through the commander chain, only at event processing time, and only carries messages about and allows responses to commands. While this form of interobject communication is vital to an application's existence, it is too specialized for general use. So instead, PowerPlant offers a broadcaster-listener mechanism for interobject communication.

A **broadcaster** is an object that sends a message when a certain behavior occurs, like when a button is clicked. A broadcaster can broadcast multiple messages and/or multiple

message types depending upon context. To make an object a broadcaster in PowerPlant, you only need to utilize the LBroadcaster mix-in class (once again, factored design) and specify where and what to broadcast. As an example, here's a fictitious button object that broadcasts when it is clicked:

Listing 3: ClickSelf for CMyButton

CMyButton::ClickSelf

```
void
CMyButton::ClickSelf(const SMouseDownEvent &inMouseDown )
{
   // the button has been clicked, notify the world
   BroadcastMessage( msg_ButtonClicked, nil);
}
```

A **listener** is an object that listens for one or more messages. When a desired message is "heard" the listener will react accordingly, perhaps beeping in response to the button click. A listener does not have to react to every message that it might receive. To make an object a listener in PowerPlant, mix in the LListener class and implement the ListenToMessage() function. Here's a fictitious listener object that is listening to the fictitious CMyButton object and beeps in response to the button being clicked:

Listing 4: ListenToMessage

CSomeObject::ListenToMessage

```
void
CSomeObject::ListenToMessage(
   MessageT    inMessage,
   void        *ioParam )
{
   // we are listening to CMyButton
   if ( inMessage == msg_ButtonClicked ) {
      ::SysBeep(3);
   }
}
```

The benefit of this factored design is that the broadcasters and listeners can remain independent of each other. The broadcasters do not need nor care to know who is listening or how they may react to the message. The listeners of course do need to know about particular messages, but they not need nor care to know about the broadcaster itself. This independence allows listeners to be added or removed at any time, for there to be any number of broadcasters or listeners (including none), and for improved object-oriented design.

Persistence

No, this does not refer to human perseverance, but rather to the need for data to exist across volatile states, such as the application launching and quitting or turning the computer on and off. This is the intent of storage media, like a hard drive or Zip disk, and the folders (to organize) and the files (to actually store the data) on that media.

The LFile class represents a basic Mac OS file (including support for both the data and resource forks) and provides the low-level means for manipulating the file and its forks (open, close, read, write, etc.). The higher-level means of file manipulation (save, save as, revert, print, etc.) are provided through the LDocument class. LDocument is an abstract class which associates one or more windows with one or more files, and also provides a means to manipulate the document (save, print, etc.) via AppleEvents. LSingleDoc is provided to give you a concrete instantiation of LDocument that actually connects an LFile to an LWindow.

If you are familiar with the C++ Standard Library, then you are probably familiar with the concept and benefits of streams (an ordered series of bytes of data). If not, to avoid a lengthy discussion here just know that streams help to transfer data from one place (file, keyboard, screen, printer) to another (file, keyboard, screen, printer). PowerPlant provides an abstract LStream class that has all the features and functions any stream class would need: marker manipulation, reading and writing data, and overloaded redirection operators to make working with various data types easier. Other provided classes are: LFileStream, for streaming data to/from a file's data fork; LDataStream, for pointer-based data; LHandleStream, for Handle-based data; and LDebugStream, for streaming information to a debugger.

Utilities

Although the main purpose of a framework is to provide a cohesive and integrated set of tools, there are always sundry tasks and chores involved in application writing. PowerPlant offers many utility classes to aid in the handling of mundane chores, keeping your code error and exception safe, and extend or even modify the behavior of your interface with a quasi-plugin architecture. It would be impossible to describe all of PowerPlant's utility features within the scope of this article, so instead here is a brief overview of some of the highlights:

- LClipboard — Manages the Clipboard.
- LGrowZone — A GrowZone implementation for managing low-memory situations.
- LPeriodical — A mechanism for calling a function on a regular basis.
- LSharable — A base class for a reference counted object.
- LString — A base class for handling Pascal-style strings.
- UDebugging — Basic debugging support.
- UDrawingState — Classes for saving/restoring the drawingstate.
- UDrawingUtils — Useful utilities for drawing (device loops, drawing text, marching ants).
- UMemoryMgr — Classes for management of memory.
- UProfiler — Aid for application profiling.
- URegions — Facilitates working with Regions.
- UTextTraits — Manages the appearance of text in UI objects.

This list is far from exhaustive; in fact many of the listed classes, like UDrawingState and UMemoryMgr, are actually files that group logical utility classes into a central location (there is no UDrawingState class, but the UDrawingState.cp file defines classes such as UQDGlobals, StGrafPortSaver, and StColorState). Take a read through the source code files and see what is offered. There is a great deal of exploration you can do and a great deal you can learn as well by reading the PowerPlant

source code. And that is another benefit of PowerPlant! Full source is provided on the CodeWarrior CD.

One area of the utility classes that I find particularly neat are the stack-based classes (these are classes that begin with "St"). The C++ language has a powerful hook: the destructor. Whenever a stack-based object goes out of scope, the destructor for that object will be called. The bonus here is that if an object exits scope normally or abnormally, say from an exception being thrown, the destructor for that object is still called. So by exploiting this hook, you can work to automate cleanup. The technique is simple: the constructor performs an action (saves a state, allocates memory, etc.) and the destructor performs the converse (restores the state, frees the memory, etc.). To illustrate this, let us look at PowerPlant's StDeleter template class. StDeleter , like the C++ Standard Library's auto_ptr, manages a pointer allocated via new. The StDeleter constructor takes ownership of the pointer, and the destructor calls delete upon it.

If you did not use StDeleter, your code might look like this:

Listing 5: Stacks without using StDeleter

```
{
  Foo  *theFoo = new Foo;
  Bar *theBar = nil;

  try {
    theBar = new Bar;
  } catch (...) {
    // Creation of theBar failed. Clean up to
    // avoid memory leaks.
    delete theFoo;

    // rethrow the exception
    throw;
  }

  try {
    theFoo->MightThrowAnException();
  } catch (...) {
    // Again, must clean up to avoid memory leaks
    delete theFoo;
    delete theBar;

    // rethrow
    throw;
  }

  delete theFoo;
  delete theBar;
}
```

As you can see, it is quite cumbersome to contend with the possibility for an exception to be thrown. We have to use numerous try/catch blocks to perform cleanups in case an exception is thrown. But if we use StDeleter to manage the objects, the code can look like this:

Listing 6: The same code as listing 5, using StDeleter

```
{
  StDeleter<Foo>  theFoo( new Foo );
  StDeleter<Bar>  theBar( new Bar );

  theFoo->MightThrowAnException();
}
```

To quote Porky Pig, "That's all folks." The design of the two code examples is the same (allocate 2 objects, call a function, avoid memory leaks, clean up after yourself), but the implementations are quite different. In this second case, if an exception is thrown when allocating theBar, theFoo's destructor is called and its memory is released. Remember, in this second example, theFoo is an StDeleter<Foo> object, not a Foo* object as it was in the first example. So when StDeleter's destructor is called, it deletes its internal pointer to the Foo* object, and all is well. If MightThrowAnException() does throw and exception, the destructors for theFoo and theBar are called and the memory is released. And if all goes without problem and we reach the end of the function, the objects leave scope normally and memory is released. Stack-based classes are not unique to PowerPlant, but PowerPlant does take advantage of this feature of the C++ language to enable you to write more robust and exception-safe code.

Conclusion

If you have made it this far, I hope that you're getting excited about PowerPlant. I have only begun to touch on the features, power, and potential of PowerPlant. There are still actions, for Undo support; AppleEvent support, to make your application fully scriptable and recordable; classes for displaying tabular data; Array classes for managing dynamic lists of data; and the list continues.

If you have questions about PowerPlant, please feel free to send me email or visit the comp.sys.mac.oop.powerplant newsgroup. Also, you might want to pull out a copy of the *PowerPlant Book*. Give a read through the first few chapters, maybe delve into later chapters and try your hand at PowerPlant coding. The next article will give you a more hands-on introduction to PowerPlant, and really show you what PowerPlant can do. Additionally, if you have any questions, comments, criticisms, or other feedback about this article or the article series, please do not hesitate to drop me a line.

Bibliography, References, and URLs

- Apple Computer, Inc. *Inside Macintosh: Overview.* Addison-Wesley Publishing Company, Reading, Massachusetts. 1992.
- Metrowerks Corporation. *PowerPlant Book.* 1998.
- Prata, Stephen. *Mitchell Waite Signature Series: C++ Primer Plus.* 3rd ed. Waite Group Press, Corte Madera, California. 1998.
- Stroustrup, Bjarne. *The C++ Programming Language.* 3rd ed. Addison-Wesley Publishing Company, Reading, Massachusetts. 1997.

<http://www.apple.com/developer/>
<http://www.metrowerks.com/>
<http://cafe.ambrosiasw.com/alt.sources.mac/>
<http://www.celestin.com/apprentice/>
<news:comp.sys.mac.oop.powerplant>
<news:comp.sys.mac.programmer.codewarrior>

by Ed Angel, University of New Mexico

OpenGL for Mac Users: Part 1

How to get started programming 3D graphical applications

Introduction

Graphics has always been one of the strongest points for the Macintosh. Although the Mac has been dominant in the graphic arts community, it has not had the same impact on the 3D CAD community or on those who write graphical applications in high-level languages. For these users, the search for a standard Application Programmer's Interface (API) has been going on for over 20 years. While official standards efforts have failed to produce an API that is actually used by a large part of the community, an API that had its origin in a particular architecture has become the choice for thousands of graphics programmers on systems ranging from SGIs to PCs to Macs.

This article will provide an introduction to OpenGL‰ We shall give the overall structure of this API and then develop a simple application program. We shall then see the ease with which we can convert our simple program to one that incorporates the dynamic and three-dimensional rendering features that we associate with modern graphics. Along the way, we shall see why this particular API should be of interest to the Mac community. A subsequent article will survey the advanced features supported by OpenGL and how OpenGL can exploit the available hardware.

The OpenGL‰ API

The search for a standard graphics API began over 20 years ago and has produced a succession of interfaces from the early CORE proposal to the ISO standard GKS and its successors (GKS-3D, PHIGS). With the possible exception of PHIGS, these APIs failed to be used by large parts of the 3D programming community. Generally, these APIs either lagged hardware developments or were not sufficiently general to be used on a wide range of graphics architectures. OpenGL‰began as GL, an API for using Silicon Graphics Inc hardware. At that time, SGI produced high-end graphics workstations that implemented much of their functionality at the chip level. Application programmers needed an API to make use of these features and GL provided such access. Somewhat surprisingly, programming in GL was considerably easier than in any of the other standard 3D APIs available at the time. OpenGL was developed as a successor to GL that could be implemented on a variety of architectures. By removing the input and windowing features of GL, it became compatible with most other windowing systems, including X Windows, Windows 95 and NT, and Mac OS. The popularity of OpenGL as a way of accessing high-level hardware features has led it to be supported by many of the add-on graphics card developers.

Figure 1 shows a typical graphics programming environment. Users write programs in a high-level language. From this perspective, the API looks like one or more libraries that contain the graphics functions. On the other end, however, the API communicates with the hardware, either directly or through a hardware abstraction layer. A good API shields the application programmers from the details of the hardware and allows them to write applications that are independent of the machine on which these applications will run. However, a good API is close enough to the hardware that it leads to efficient use of these resources and allows the application programmer to make use of hardware features.

Ed Angel is a Professor of Computer Science and Electrical and Computer Engineering at the University of New Mexico. He is the author of Interactive Computer Graphics: A top-down with OpenGL (Addison-Wesley, 1997). You can find out more about him at www.cs.unm.edu/~angel or write him at angel@cs.unm.edu.

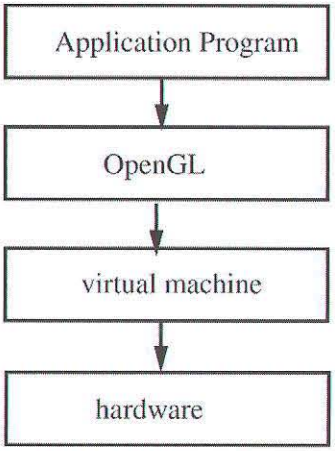


Figure1. *Levels of a Graphics Environment.*

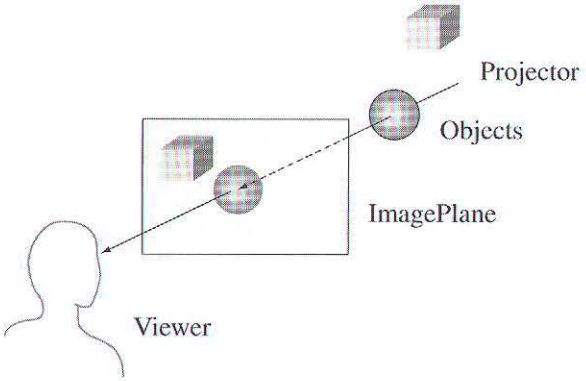


Figure2. *Synthetic Camera.*

Most modern graphics systems are based upon the synthetic camera model in **Figure 2.** A program must allow the user to specify the objects in the scene and the viewer (or camera). Because both objects and viewer are in a three-dimensional world, the API should allow their specification in a three-dimensional coordinate system. The details of how these specifications are combined mathematically, via the projection process, to form a two-dimensional image is the job of the graphics hardware and software.

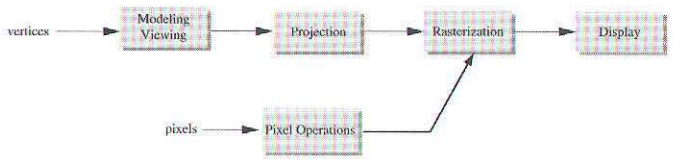


Figure 3. *Graphics Pipeline.*

Graphics architectures, whether workstations or special purpose boards, have evolved to the pipeline configurations similar to the one in **Figure 3.** Graphical objects are specified through a set of locations in space (vertices) that are passed through a first transformation that scales, orients, and positions the objects and a second that projects the resulting positions to their correct two-dimensional locations on the screen. The collection of parameters necessary to accomplish these tasks — rotation angles, scaling coefficients, camera angles — are part of the state of the graphics system. As graphical objects flow down the pipeline, the present state determines how they are to be rendered; whether they should be shaded, whether hidden-surface removal should take place, and whether they should be texture mapped. Because many of the high level functions that users now expect, such as texture mapping and compositing, involve the use of arrays of picture elements (pixels), modern architectures also provide a parallel pipeline for pixels and access to a variety of buffers.

The OpenGL API is based on this architecture. We can divide the routines into a few basic categories

- *Primitive specification functions* that defined the atomic objects that OpenGL supports. These include geometric primitives, such as line segments and polygons, and discrete primitives, such as bitmaps, and pixel arrays. Geometric primitives are supported in two, three and four dimensions and in most of the standard data formats. Higher level objects including Bezier curves and surfaces, quadrics, and NURBS curves and surfaces are supported, although are rendering through the simpler primitives.

- *Attribute specification functions* set colors, line types, fill patterns, material properties and lighting specifications.

- *Transformation functions* control both geometric transformations (rotation, translation, and scaling) and the viewing transformations (orthographic and perspective).

- *Control functions* enable various features including hidden-surface removal, lighting, texture mapping, blending and antialiasing.

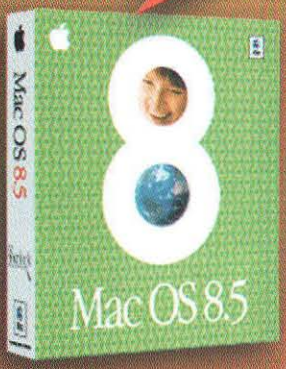
These functions generally are contained in two libraries: GL which contains the basic functions, and GLU, which contains useful additional functions built from functions in the GL library. OpenGL is a rendering library whose main concern is displaying graphical entities on the screen. It does not contain windowing and input functions. This lack allows us to use OpenGL with a variety of systems but allows OpenGL applications to make use of the local environment. Most implementations provide a basic library of interface routines. For the Mac, these routines are in a library called AGL. One alternative is a simple library called GLUT (OpenGL Utility Toolkit) that incorporates the functionality common to the standard windowing systems and has been implemented on most of the standard systems. GLUT allows us to put a window on the display and get input from the mouse and keyboard through callback functions that define how OpenGL should react to events within the windowing environment. Because in this article we are concerned with the basics of OpenGL and portability, we shall make use of the library.

Best Show Specials!

These products and hundreds more!

Visit us at

www.devdepot.com



Mac OS 8.5 ONLY $85!

by Apple Computer, Inc.

Upgrade to the most powerful version of the Macintosh OS yet! Mac OS 8.5 offers faster file transfer, easier internet connectivity, and the powerfull search capabilities of Sherlock!

SPECIAL LIMITED TIME ONLY PRICE! (only while current supplies last)

PowerKey Rebound! ONLY $89!

by Sophisticated Circuits

Rebound! is an ADB crash recovery system for the Macintosh. Its software monitors the system for crashes and restarts it automatically by sending a code from Rebound!

Resorcerer 2.2 ONLY $256!

by Mathemaesthetics

The premiere Macintosh resource editor. Edits over 3 times as many resource types as ResEdit — including Alias resources and a new filtered TMPL for Mac OS 8.5 'icns' (Icon Suite) resources!

ChemOffice $289

Scripter 2.0 $199

WebTen $485

VOODOO $199

FaceSpan 3.0 $189

Open GL $379

Adobe PageMill 3.0 $94

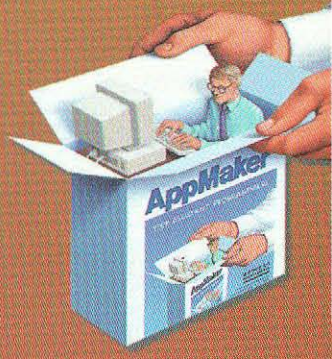
ToolsPlus Lite $99

Developer DEPOT®

PO Box 5200 • Westlake Village, CA • 91359-5200 • Voice: 800/MACDEV-1 (800/622-3381)
Outside US/Canada: 805/494-9797 • Fax: 805/494-9798 • E-mail: orders@devdepot.com

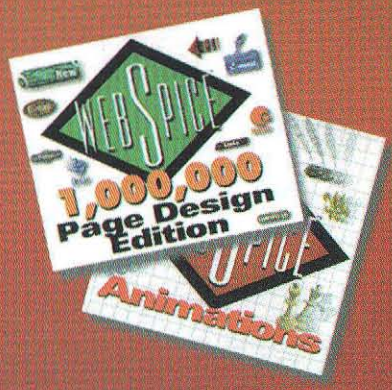
MacWorld EXPO SAN FRANCISCO 1999

The products you need, with the prices and service you deserve... guaranteed.

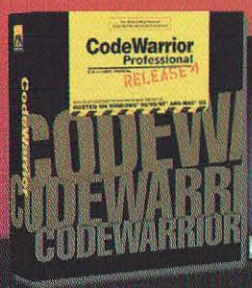


AppMaker ONLY $199!
by Bowers Development
Create your applications user interface with drag and drop, point and click ease!

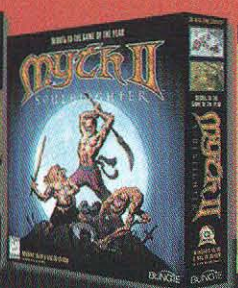
Funnel Web Professional ONLY $249!
by Active Concepts
Professional level web site analysis and profiling. Provides all the features of the standard version with the additional support for:
- Professional level engine for high server loads Virtual domain Proxy Analysis
- Realtime updating Clickstream Analysis.



WebSpice Animation ONLY $99!
WebSpice 1,000,000 Page Design Edition ONLY $99!
by DeMorgan Industries Corporation
Create with royalty-free graphics and animation with 6 CD's of high quality web graphics or 2 CD's of web animation!



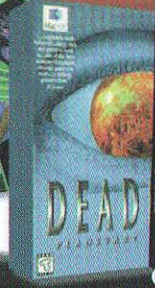
CodeWarrior Pro 4
$379



Myth II Soulblighter
$43.95

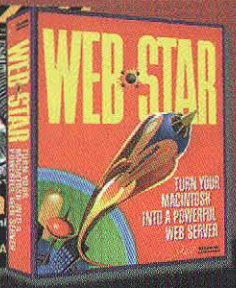
GoLive CyberStudio 3.1
$289

CoolDVD
$449



Deadlock
$46.95

CodeWarrior
Discover Programming
$69



WebSTAR
$489

Program Structure

A typical OpenGL application consists of a set of functions including

- main which initializes the system, describes the required display properties, sets up any menus, and names the callback functions. The last statement of a typical main function puts the program in an event loop and the program's further behavior is determined through the callbacks.

- display is a callback that is invoked whenever the windowing system determines the OpenGL window must be redisplayed. One time is when the window is first opened. Subsequent calls to this callback are made from other callbacks.

- myinit is a user defined function that includes atttributes and enabling of OpenGL capabilities. A user may decide to put menu definitions here rather than in the main function.

These three functions will appear in virtually all OpenGL programs. Most applications will make use of at least a few other callbacks. In GLUT, these include a mouse callback, a keyboard callback, an idle callback that is used whenever no other events are pending, and a reshape callback that is called whenever the window is resized.

A Simple Application

Our first example draws a box on the display once. It uses the GLUT library to define a screen window of 200 x 200 pixels.

Listing 1: box.c

```
/* Display a two-dimensional box in OpenGL */

#include "glut.h"                    /* should include gl.h and glu.h */

display()
{
  glBegin(GL_LINE_LOOP);             /* Square centered at origin */
    glVertex2f(-1.0, -1.0);
    glVertex2f(-1.0, 1.0);
    glVertex2f(1.0, 1.0);
    glVertex2f(1.0, -1.0);
  glEnd();
}

void myinit()
{
  glColor3f(1.0, 1.0, 1.0);                /* Drawing color */
  glMatrixMode(GL_PROJECTION);             /* Set clipping window */
  glLoadIdentity();
  gluOrtho2D(-2.0, 2.0, -2.0, 2.0);
  glClearColor(0.0, 0.0, 0.0, 1.0);        /* set clear color */
  glClear(COLOR_BUFFER_BIT);               /* now clear Frame Buffer */
}

void main(int argc, char **argv)
{
  glutInit(&argc, argv);              /* Initialize GLUT */
  glutInitDisplayMode(GLUT_RGB);      /* Request 3 color screen window */
  glutInitWindow(200, 200);           /* request 200 x 200 window */
  glutCreateWindow("box");            /* Window title */
  glutDisplayFunc(display);           /* Name display callback */
  myinit();                           /* User settings */
  glutMainLoop();                     /* Enter event loop */
}
```

Although there are six GLUT functions in our initialization, these will be almost identical in most applications. In fact, there are default settings that would have been used if we had omitted glutInitDisplayMode, glutInitWindow, glutCreateWindow and the setting of colors. The display callback requires only the name of the function that will be run and we can place most of our drawing functions in this function. Note the form of the specification of the square. We state what we want to draw in glBegin, follow it with a list of vertices, and finish with a glEnd. There are multiple forms of glVertex that allow us to use the standard data types, such as int (i), float (f) and double (d). We can specify vertices in 2, 3 or 4 dimensions and either specify the components or, as we shall see in the next example, point to an array of the components. Strings such as GL_LINE_LOOP are defined in gl.h, which is normally included by glut.h.

OpenGL supports three-color (RGB) , four-color (RGBA) and color-index mode graphics with RGB as the default. Colors that are specified as floats or doubles range from 0.0 to 1.0 where 0.0 means none of that component and 1.0 means fully on. For integers and unsigned bytes, color components range over the full range of the type. The clear color is in RGBA format and in our examples clears the screen to black and opaque.

We specify a two-dimensional rectangle as a line loop in which the final vertex specified is automatically connected to the first. The code in our display callback only describes the object and not the camera. In OpenGL, there is a default camera at the origin pointing in the negative z direction. This camera only sees the region in a 2 x 2 x 2 box centered at the origin. Objects outside this region are clipped out. Most applications require more flexibility so we specify a camera and clipping region through the projection matrix. In two dimensions, the function gluOrtho2D forms an orthogonal projection matrix and sets a clipping rectangle in the two-dimensional object plane that is mapped to the screen window specified by glutInitWindow. The matrix becomes part of the graphics state. We set it by specifying which matrix we want to change (glMatrixMode), intializing it to an identity matrix (glLoadIdentity) and multiply the identity matrix on the right by the projection matrix (glOrth2d).

A Rotating Cube

Our second example is three dimensional and includes some dynamic behavior. A cube has eight vertices and six faces. We can build the cube from six squares through a function quad.c that specifies a quadrilateral through pointers to a global array of vertices, such as

```
GLfloat vertices[8][3] = {{-1.0,-1.0,-1.0}, {1.0,-1.0,-1.0},
  {-1.0,1.0,-1.0}, {1.0,1.0,-1.0},{-1.0,-1.0,1.0},
  {1.0,-1.0,1.0},{-1.0,1.0,1.0},{1.0,1.0,1.0}};
```

To color the cube we specify a global color array

```
GLfloat colors[8][3] = {{0.0, 0.0, 0.0}, {1.0, 0.0, 0.0},
  {0.0, 1.0, 0.0}, {1.0, 1.0, 0.0}, {0.0, 0.0, 1.0},
  {1.0, 0.0, 1.0}, {0.0, 1.0, 1.0}, {1.0, 1.0, 1.0}};
```

Note our use of built-in types (defined in gl.h), such as GLfloat instead of float.

Listing 2: quad.c

```
void quad(int a, int b, int c, int d)
{
    glBegin(GL_QUAD)
      glColor3fv(colors[a]);
      glVertex3fv(vertices[a]);
      glVertex3fv(vertices[b]);
      glVertex3fv(vertices[c]);
      glVertex3fv(vertices[d]);
    glEnd();
}
```

In OpenGL, colors are part of the state and as each vertex travels down the pipeline it is assigned the present color. Our quad function assigns the same color to each vertex causing the entire object — a filled quadrilateral — to have the same color. If we were to assign a different color before each call to glVertex, each could have a different color and OpenGL would interpolate the colors at the vertices across the quadrilateral. A cube can be constructed from the quadrilaterals through the function callback cube.c. Note that because the vertices are numbered as in Figure 4, the order of the indices in our calls to quad conform to the right-hand rule, that is if we look at a face from the outside then we follow the vertices in a counter clockwise manner. Although this ordering will not affect our simple example, it is crucial to maintain a consistent order when we add lighting to our example.

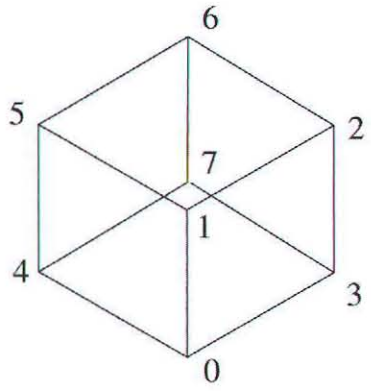


Figure 4. *Cube Vertex Labeling.*

Listing 3: cube.c

```
void cube()
{
  quad(3, 1, 0, 2);
  quad(2, 6, 7, 3);
  quad(6, 2, 0, 4);
  quad(1, 3, 7, 5);
  quad(4, 5, 7, 6);
  quad(5, 4, 0, 1);
}
```

We can use almost the same main program as before but we want to use hidden-surface removal so that points that should not be visible because there is a surface between them and the camera will not be displayed. OpenGL has a particular hidden-surface algorithm — the z-buffer algorithm — built in. It uses a special buffer called the depth buffer that must be requested at initialization (and supported by the implementation) and the process must be enabled, either in myinit or in main. Our plan for this program is that as long as nothing else is happening the cube will rotate about one of the coordinate axes, which axis will be selected by a three–button mouse (or one emulated with one-button mouse plus keys). Consequently, we shall need both a mouse callback and an idle callback . For smooth display of our rotating cube we request double buffering. In this mode, OpenGL will always display the front buffer and draw into the back buffer. The application program can clear and swap these buffers as necessary. Finally, we use a keyboard callback so that hitting the q or Q key will terminate the program.

Listing 4: main.c

```
void main(int argc, char **argv)
{
  glutInit(&argc, argv);              /* Initialize GLUT */
  glutInitDisplayMode(GLUT_RGB | GL_DOUBLE | GL_DEPTH);
                    /* Request 3 color window, z buffer & double buffering */
  glutInitWindow(200, 200);
                    /* request 200 x 200  initial  window */
  glutCreateWindow("box");            /* Window title */
  glutDisplayFunc(display);           /* display callback */
  glutIdleFunc(idle);                 /* idle callback */
  glutMouseFunc(mouse);               /* mouse callback */
  glutKeyboardFunc(mykey);            /* keyboard callback */
  glutMainLoop();                     /* Enter event loop */
}
```

The idle callback is executed whenever there are no other events are pending. Here we simply increase the angle of rotation by a fixed amount and call the display function.

Listing 5: idle.c, myinit.c, mykey.c and display.c

```
float theta[3];                    /*The 3 angles of rotation */
static int axis = 0;               /* which axis to rotate about */

void idle()
{
  theta[axis] += 2.0;              /* increment angle by 2 degrees */
  if(theta[axis] > 360.0) theta[axis] -= 360.0;
  glutPostRedisplay();
}

void mykey(unsigned char key, int mousex, int mousey)
{
  if((key=='q')||(key=='Q')) exit(); /* terminate program */
}

void display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                                     /* clear frame buffer and z buffer */
  glLoadIdentity();
  glRotatef(theta[0], 1.0, 0.0, 0.0);    /* x rotation */
  glRotatef(theta[1], 0.0, 1.0, 0.0);    /* y rotation */
  glRotatef(theta[2], 0.0, 0.0, 1.0);    /* z rotation */
  cube();
  glutSwapBuffers();                     /* double buffering */
}

void myinit()
{
  glEnable(GL_DEPTH_TEST);           /* Enable hidden-surface removal */
  glMatrixMode(GL_PROJECTION);       /* set viewing box 2 x 2 x 2 cube */
  glLoadIdentity();
  glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
  glMatrixMode(GL_MODELVIEW);        /* return to modelview matrix */
}
```

There are few subtleties in this code. We now set up the desired clipping window in myint using glOrtho to define a three-dimensional clipping box for an orthogonal projection. If we prefer a perspective projection we can use either glFrustum or gluPerspective. Note that after we set the viewing conditions we change the matrix mode so that later transformations in the display callback will be applied to the modelview matrix and not to the projection matrix. Because the modelview matrix is part of the state, it must be set before we send the vertices down the pipeline. We use the three rotation angles to define separate rotations about the x, y and z axes. The mouse callback allows the user to change which of these angles is incremented by the idle callback. Note that because transformation matrices are multiplied on the right of the existing modelview matrix, the order of the code in our example causes the rotation about the z axis to be done first, followed by the rotation about the y axes and finally by the rotation about the x axis. We invoke the display callback indirectly through glPostRedisplay rather than display so that, if possible, the implementation can reduce the number of screen redraws.

Smooth Shading and Transparency

At this point, we have a working program that illustrates many OpenGL features. But we are just getting to the good part. Additional capabilities are only a few lines of code away. Let's demonstrate two simple changes: smooth colors and transparency. Colors are assigned to vertices. Because color is part of the graphics state, if we only assign a color before the first vertex, the color will be used for all the vertices and this color becomes the color of the face. If we assign colors to all the vertices, the vertex colors will be interpolated over the polygon as it is rendered. We now assign the four colors in quad

Listing 2: revised quad.c

```
void quad(int a, int b, int c, int d)
{
     glBegin(GL_QUAD)
       glColor3fv(colors[a]);
       glVertex3fv(vertices[a]);
       glColor3fv(colors[b]);
       glVertex3fv(vertices[b]);
       glColor3fv(colors[c]);
       glVertex3fv(vertices[c]);
       glColor3fv(colors[d]);
       glVertex3fv(vertices[d]);
     glEnd();
}
```

Even if colors are specified for all the vertices, we can still have the first vertex color determine the color of the entire polygon by specifying flat shading by

```
GlShadeModel(GL_FLAT);
```

The default is smooth shading (GL_SMOOTH). The difference between flat and smooth shading is shown in **Figure 5.**

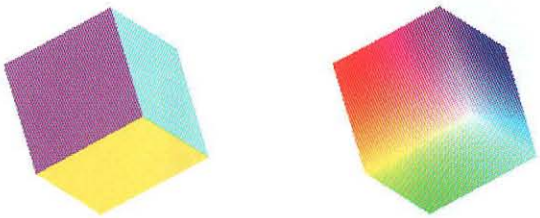


Figure 5. *Flat and Smooth Shaded Cubes.*

Figure 6. *Translucent Cube.*

Transparency relies on a four-color RGBA model. The fourth "color" is called the *alpha* component and is used to represent the transparency of the point to be colored, with 0.0 meaning fully transparent and 1.0 meaning completely opaque. The alpha value is handled as the other color components and is assigned to a vertex. If smooth shading is enabled, the alpha value will also be interpolated across the

polygon. Transparent polygons are blended together. We can enable blending by

```
GlEnable(GL_BLEND);
```

We must also decide how polygons should be blended into the frame buffer. The most common method is to use the alpha of the polygon being rendered (the source alpha) to multiply the RGBA color of the point being displayed, as computed from the vertex colors, and to combine this color with the color already in the frame buffer (the destination color) scaled by one minus the source alpha. We set this mode in myinit by

```
GlBlendingFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The order in which polygons are rendered affects the appearance of a blended display. In addition, hidden-surface removal combined with blending can cause some odd results and we usually disable the depth buffer if we want simple blending. The image in **Figure 6** was created using the color array

```
GLfloat colors[8][4]={{0.0,0.0,0.0,0.5},{1.0,0.0,0.0, 0.5},
  {0.0,1.0,0.0,0.5}, {1.0,1.0,0.0,0.5}, {0.0,0.0,1.0,0.5},
  {1.0, 0.0,1.0,0.5},{0.0,1.0,1.0,0.5},{1.0,1.0,1.0,0.5}};
```

and changing each glColor3fv in quad to a glColor4fv.

What Comes Next

Many further embellishments are easy. We could modify the keyboard callback to enable and disable all the features we have discussed or to turn off rotation by setting the idle callback function to NULL. In the second article, we shall add lights and shading. We will do so by replacing colors with material properties, adding orientation to the vertices, and enable lighting and one or more light sources. We will also map images (texture maps) to our objects.

On the input side we have two choices. If we want to develop cross-platform code, we can continue with GLUT and exploit other features including additional callbacks, menus, and multiple windows. Or we can use the native environment on the Mac. By concentrating on rendering, OpenGL is compatible with the underlying OS and its input and windowing functions.

We shall also look at the pixel processing pipeline and the use of OpenGL buffers for a variety of tasks such as manipulating bitmaps, creating stereo images, and antialiasing. We shall also look at how OpenGL can make use of advanced hardware features and how application programmers can optimize graphics performance.

Conclusions

On the personal side, I have used various graphical APIs in my own work and teaching for the last 15 years. Although I was supportive of standards, the reality was usually far below my hopes. When some of my students got me to try OpenGL in my classes a few years ago, I was skeptical. However, the results were nothing short of astounding. Not only could all the students produce three-dimensional applications after a short time, they worked on Macs, PCs, SGIs and a variety of other systems, with no thought for which platform they were writing code.

In this article, we have focussed on introducing the basic features of the OpenGL API. At this point, armed with a handful of OpenGL functions, you should be able to write some serious applications. In our second article, we shall introduce more advanced features of the API. Besides being an easy API with which to write graphics code, the design of the OpenGL API is close enough to the hardware that applications run efficiently. That combination of features plus portability should make OpenGL attractive to Mac users.

Sources and URLs

OpenGL is administered through an Architectural Review Board. The two major sources for on-line information on OpenGL are the OpenGL organization <http://www.opengl.org> and Silicon Graphics Inc <http://www.sgi.com/Technology/OpenGL>. You can find pointers to code, FAQ, standards documents and literature at these sites. I keep the sample code from my book at <ftp://ftp.cs.unm.edu under pub/angel/BOOK>.

OpenGL is available for most systems. For Mac users, there is an implementation from Conix Enterprises <http://www.conix3d.com> that includes support for GLUT and for hardware accelerators. There is a free OpenGL-like API called Mesa <http://www.ssec.wisc.edu/~brianp/Mesa.html> that can be compiled for most systems, including linux, and will run almost all OpenGL applications. You can obtain the code for GLUT and many examples at <http://reality.sgi.com/opengl/glut3/glut3.html>.

The standard OpenGL references are the Reference Manual and Programming Guide and My book is an introductory computer graphics textbook that uses OpenGL. Kilgard contains information on GLUT and interfacing OpenGL to window systems. Although Wright and Sweet are oriented towards PC implementations of OpenGL, their book contains many introductory examples that are platform independent.

Bibliography and References

- Angel, Edward. *Interactive Computer Graphics: A top-down approach with OpenGL.* Addison-Wesley, Reading, MA, 1997.
- Arhictectural Review Board. *OpenGL Reference Manual, Second Edition,* Addison-Wesley, 1997.
- Kilgard, Mark, *OpenGL Programming for the X Window System,* Addison-Wesley, 1996.
- Neider, Jackie, Tom Davis and Mason Woo. *The OpenGL Programming Guide, Second Edition.* Addison-Wesley, Reading, MA, 1997.
- Wright, Richard Jr. and Michael Sweet, *The OpenGL Superbible,* Waite Group Press, Corte Madera, CA, 1997.

Acknowledgements

I would like to thank Apple Computer, Silicon Graphics, and Conix Enterprises for the hardware and software support that enabled me to write my OpenGL textbook.

MT

by Kas Thomas

Poor Man's Bryce, Part III: Faster Terrains in QuickDraw 3D

Follow a few simple tips and you're guaranteed to get better performance from QuickDraw 3D

Apple's QuickDraw 3D library offers graphics programmers a powerful, flexible API for implementing 3D graphics in a cross-platform manner. Using QD3D, you can get interactive 3D graphics to happen on screen with relatively minimal effort. The important concept here is "interactive": From the outset, QD3D's designers wanted interactivity to be part of the 3D experience, which means the API has been painstakingly optimized for speed. Unless you're modeling truly enormous objects (i.e., with tens of thousands of polygons), almost anything you create with QD3D can be manipulated onscreen (i.e., rotated, scaled, translated) in real time, fully textured and smooth-shaded. This is far different from most raytracing environments, where you wait from a few seconds to several hours for a single screen to draw.

But in QD3D programming — as in other types of graphics programming — you can never have too much speed. It takes so much floating-point math to render a scene (even if you're just doing shadowless, reflectionless flat-shaded objects) that it's not hard at all to run into situations where there is a noticeable, annoying lag between mouse movements and screen updates, even on a G3 machine with video acceleration.

In previous articles (*MacTech*, October and November 1998), I showed how to put together a simple QD3D program, called PMB (for "Poor Man's Bryce"), that can convert 2D imagery to 3D "terrain" using displacement mapping. (The full Code Warrior projects are available online at ftp.mactech.com.) In Part I, we set up the code to make a height-mapped 3D grid with vertices equal (or at least proportional) in number to the number of pixels in the source image; and we presented code to render this grid in wireframe, dots, or flat-shaded mode. The terrain objects we made this way weren't particularly attractive, so in Part II we looked at some techniques for "prettying up" our terrains. We talked, for example, about how to smooth out the facets of the terrain by means of *vertex-normal recalculation* (giving true Gouraud shading); and we showed how to overlay our grids with PICT images (honest-to-gosh *texture mapping*). Surprisingly, we didn't take much of a speed hit along the way. It turns out, for example, that by precalculating our vertex normals (in preparation for Gouraud shading), we save QD3D's rendering engine from having to calculate normals on the fly, which it would otherwise do.

Still, it would be nice if our terrains rendered a bit quicker, so that large terrains (with, say, 10,000 polygons) could be swiveled, resized, etc. in snappier, more "interactive" fashion. As it is, version 2.0 of our PMB app requires 25 ticks (about four tenths of a second) to rotate an 8,000-polygon object 12 degrees around the y-axis, and display it fully updated. This is for a fully textured (with a 360-by-360-pixel texture map) TriGrid, drawn into a 480x384 window in 32-bit color, on a 250Mhz G3 machine. Mind you, that's two and a half frames per second, which for some types of work is not so bad. Still, it's a long way from being realtime-interactive.

I promised last time that there would be ways to speed up our code significantly. It's time now to deliver on that promise. In the pages to follow, we'll see how it's possible to achieve a better than five-to-one speedup of our program, with no loss of functionality. Many of the techniques we'll discuss can be applied to other QD3D programs, for similar speed gains. So fasten your seat belt, and get ready to rocket.

Kas Thomas <tbo@earthlink.net> has been programming in C on the Mac since 1989. He is the author of a QD3D-powered Photoshop® plug-in called Callisto3D (available for download, free, at http://users.aol.com/callisto3d), which uses many of the techniques discussed in this article.

Choice of Geometry

Let's get right to the core of the matter and start with one of the most important speed considerations, namely choice of geometry. As mentioned in Part 1 of this series, QuickDraw 3D offers four freeform mesh primitives that can be used to represent complex objects. They include the *Mesh*, the *Trigrid*, *Polyhedron*, and *TriMesh*. (Of course, QD3D also accommodates NURB patches, but that's another story.) By way of review, here are the main distinguishing features of these geometries:

- *Mesh:* This was the original "complex geometry" that shipped with Version 1.0 of QuickDraw 3D. It's by far the most flexible geometry, because it allows you to specify non-triangular polygons, with "holes" cut in them if you desire, and you can attach any number of attributes (in any combination) to any of a mesh's vertices, edges, faces, contours, or component groupings. But precisely *because* of the mesh's flexibility and generality, it is the slowest-rendering freeform primitive. The mesh, by its nature, brings with it a lot of code overhead at render-time. It is the worst choice of primitive where speed is concerned.

- *TriGrid:* This type of object differs from the Mesh in that it comprises a lattice of connected triangles, with equal numbers of vertices in each row and equal numbers of vertices in each column. Efficient sharing of vertices by neighboring triangles makes the TriGrid a relatively efficient (fast-rendering) object. It happens to be well-suited to our terrain-generation task, since adjoining pixels in a 2D source image can easily be mapped to adjoining vertices in a TriGrid. For more complex modelling tasks, however, it is obviously somewhat limited, since not every 3D object lends itself to a fixed-lattice layout.

- *Polyhedron:* The Polyhedron uses a triangle list defined by indexes into a vertex list. In other words, it's essentially an arbitrary array of triangles. The triangles may share vertices, or they may not — it's totally up to the programmer. This is a more orthodox type of "freeform mesh" object, extremely flexible topologically, yet relatively efficient in terms of RAM requirements and rendering speed. Unlike the Mesh, the Polyhedron can't contain polygons with more than three vertices, nor can it contain nested "component" regions or cutouts. But the Polyhedron's fast rendering speed moots most such concerns. It's an efficient, easy-to-work-with primitive.

- *TriMesh:* The TriMesh is by far the fastest-rendering of QD3D's freeform primitives (as we'll see in a moment). Structurally, it's a lot like the Polyhedron in that it is essentially an array of triangles defined by indices into an array of points. But there are some key differences. Whereas the Polyhedron follows the QD3D tradition of allowing attributes to be individually assigned to triangles, edges, and/or vertices, as well as the whole object, the TriMesh imposes a "uniform attributes" requirement, such that if *one* triangle has a transparency attribute (for example), *all* triangles have to have a transparency attribute. That doesn't mean that you can't "nil out" the transparencies of those triangles you don't *want* to be transparent, but you *do* have to specify attribute storage for all triangles, regardless of what the attribute *value* is for each one.

The TriMesh is like a Polyhedron in a straightjacket. It lacks some of the flexibility of the Polyhedron (and other QD3D mesh primitives), and for that reason it is — not surprisingly — a lot better-performing. Simply put, there is very little "special case" code overhead for the TriMesh. The renderer knows in advance what to do to make the TriMesh show up onscreen. It doesn't have to stop and examine every attribute of every face and vertex, because if a given attribute type is present for *one* face or vertex, it's going to be present for *all*. Some very good rendering optimizations result.

The TriMesh is a "low level," performance-optimized object. Like many low-level tools, it's a bit harder (and less forgiving) to use than the higher-level, "programmer-centric" primitives. It takes some getting used to. But if your primary need is speed, this is one animal you'll definitely want to spend time getting to know.

Conversion Code

One strategy that's worth considering (for almost any QD3D project) is using two versions of a given object: an easy-to-code, offscreen, "working" version, and a render-time version. The behind-the-scenes "working" version of the object might be a TriGrid or Mesh, while the renderable version might be a TriMesh (for speed). All you need is a routine that can convert from one version to the other. This is what I did for Version 3.0 of our sample app, PMB (code available online). I added a menu option under the Edit menu called "Swap Geometries," and when the user chooses this item, a routine named DoTriMeshConversion() translates our TriGrid to a TriMesh (if it hasn't been made already). In this way, the user can toggle back and forth between TriGrid and TriMesh versions of the terrain very quickly.

At this point it might be a good idea to review the data structure describing a TriMesh.

```
typedef struct TQ3TriMeshData {

   TQ3AttributeSet              TriMeshAttributeSet;

   unsigned long                numTriangles;
   TQ3TriMeshTriangleData       *triangles;

   unsigned long                numTriangleAttributeTypes;
   TQ3TriMeshAttributeData      *triangleAttributeTypes;

   unsigned long                numEdges;
   TQ3TriMeshEdgeData           *edges;

   unsigned long                numEdgeAttributeTypes;
   TQ3TriMeshAttributeData      *edgeAttributeTypes;

   unsigned long                numPoints;
   TQ3Point3D                   *points;

   unsigned long                numVertexAttributeTypes;
   TQ3TriMeshAttributeData      *vertexAttributeTypes;

   TQ3BoundingBox               bBox;

} TQ3TriMeshData;
```

The code to translate our TriGrid to a TriMesh is, frankly, somewhat lengthy and ugly. It comprises a separate (new) C module in our project, TriMeshConversion.c, which contains roughly

300 lines of code divided among five routines. In terms of creating the TriMesh, the main thing to keep in mind is that it's almost always possible to "nil out" many of the TQ3TriMeshData structure's fields. For example, in **PMB**, we're not applying an attribute set to the overall object, hence we can set the triMeshAttributeSet field to nil. We also don't want to specify any triangle attributes, per se, nor edges (nor edge attributes), so the relevant fields are all zeroed out. (Frankly, if you start to use very many of these fields, the speed advantage of the TriMesh quickly begins to evaporate.) We *do* have to specify a point list, of course, as well as a triangle list consisting of indices into the point list, because this is how we describe our geometry. The TriMesh also requires that we precalculate a *bounding box* for the bBox field (which helps speed rendering). This is not hard to do: You simply loop over all of the points in the TriMesh and find the minimum and maximum x,y, and z coordinates of the points. It's very important to do this correctly, however, because if you don't, you'll crash your computer.

Transcribing our geometry information from TriGrid form to TriMesh form is a snap. **Listing 1** shows how we copy our point list data.

Listing 1: ConvertVertices()

ConvertVertices()

Copy point list from trigrid into trimesh data.

```
void ConvertVertices( TQ3TriGridData *tgData,
          TQ3TriMeshData *tm) {
  long i;

  tm->numPoints = tgData->numColumns * tgData->numRows;
  tm->points = (TQ3Point3D *)NewPtr(sizeof(TQ3Point3D)
                                    * tm->numPoints);

  for (i = 0,tm->points != nil; i < tm->numPoints; i++)
    tm->points[i] = tgData->vertices[i].point;
}
```

The number of points is just the number of rows of the TriGrid times the number of columns. We allocate an appropriate amount of memory, then loop over all the vertices in the TriGrid, copying point information directly into the new array. A piece of cake.

It's not at all hard, either, to copy triangle data from a TriGrid to a TriMesh (or to a Polyhedron). **Listing 2** shows how it is done.

Listing 2: ConvertGridTriangles()

ConvertGridTriangles ()

```
void ConvertGridTriangles( TQ3TriGridData *tgData,
          TQ3TriMeshData *tm,
          unsigned long *numTriangles)
{
  unsigned long col,row,count,numCols,numRows;
  unsigned long triangle_set = 0;
  TQ3TriMeshTriangleData *ptd;

  ptd = (TQ3TriMeshTriangleData *) NewPtr(
      (tgData->numColumns-1) *
      (tgData->numRows-1) *
      2 * sizeof( TQ3TriMeshTriangleData ) );

  if (ptd == nil) return; // failure? go back
  numCols = tgData->numColumns;

  numRows = tgData->numRows;

  for (count = row = 0; row < numRows - 1; row++)
    for (col = 0; col < numCols - 1; col++)
    {

      // we'll assume an ordering
      // of pts clockwise from upper left

      // first do 1-2-4
      ptd[ count ].pointIndices[0] =
        col + row * numCols;
      ptd[ count ].pointIndices[1] =
        col + 1 + row * numCols;
      ptd[ count ].pointIndices[2] =
        col + numCols + row * numCols;
      count++;

      // then 4-2-3
      ptd[ count ].pointIndices[0] =
        col + numCols + row * numCols;
      ptd[ count ].pointIndices[1] =
        col + 1 + row * numCols;
      ptd[ count ].pointIndices[2] =
        col + 1 + numCols + row * numCols;
      count++;

    } // end of double nested loop

  *numTriangles = count;

  tm->triangles = ptd;
}
```

Since the TriGrid is a rectilinear lattice, you can just raster through the rows of points, forming triangles as you go. (The triangles are just triples of indexes into the array of points: i.e., three point-array indices define a triangle.) In **Listing 2**, we follow a point-traversal scheme based on the ordering shown in **Figure 1**. You can switch hypotenuse directions within rows and/or across columns, if you want to get creative, but you should specify points in a consistently clockwise (or consistently CCW) order, if you want to avoid "flipped polygons" at render time.

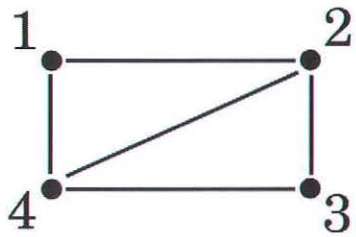


Figure 1.

When triangulating the cells of a TriGrid, it's possible to go in any direction. For our TriGrid-to-TriMesh conversion routine, we chose to copy points in 1–2–4, 4–2–3 order. (Notice that a clockwise orientation is thus maintained.) In this instance, the hypotenuse runs from lower left to upper right, but it's just as easy to have the hypotenuse go from upper left to lower right. You can also vary the orientation of the hypotenuse from one cell to the next, or from one row to the next.

Attributes

The tricky part of setting up any TriMesh involves attributes. You may have noticed, back in the code for **Listing 1**, that a TriGrid's triangles are based on vertices (type TQ3Vertex3D), whereas a TriMesh's triangles are based on points (type TQ3Point3D). The difference is that a TQ3Vertex3D is a structure that encapsulates both a *point* and an *attribute set*, whereas points don't have attributes. With a TQ3Vertex3D, you can easily

attach various attributes to particular points; this is in keeping with the object-oriented nature of QD3D. (Attributes are bound tightly to the objects they modify.) The TriMesh way of doing things, by contrast, is to specify a *point list* in one array, and a corresponding *attribute list* in a separate (but equal) array.

In our case, we happen to have a terrain object in which individual vertices may or may not have one (or more) of three kinds of attributes: vertex normals, diffuse colors, and/or "UV" (parametric mapping) coordinates. Vertex normals enable us to get a smoothed, Gouraud-shaded object (without ugly faceting). Diffuse colors give each vertex an RGB value. UV parameters are what let us do texture mapping. (For a review of these subjects, see last month's article.) Note that vertex normals are 3D vectors, whereas diffuse colors are given by three floats and UV parameters are pairs of floats.

To get our vertex attributes into a form that the TriMesh can use, we have to loop across all of the vertices in the original TriGrid and call Q3AttributeSet_Get on each vertex. This call plucks the actual *data* out of the attribute set for each vertex. If you specify kQ3AttributeTypeNormal as an argument, you'll get vector data back. If you specify kQ3AttributeTypeShadingUV, you'll get the TQ3Param2D for the vertex (describing its UV coordinates). But the point is, the TriMesh expects to have *contiguous attribute data*, which is to say, if there are 1,000 points in a mesh, and they all have vertex normals as well as UV params, you'll need to specify an array of 1,000 vectors *and* a second array of 1,000 UV params. If the points just have UV params but no other attribute types, then you just need to form an array of UV params.

A TriMesh expects its attribute data to be given in a particular format, as shown below:

```
typedef struct TQ3TriMeshAttributeData
{
  TQ3AttributeType   attributeType;
  void               *data;
  char               *attributeUseArray;
}
```

The first field of this structure will be set to a predefined constant, like kQ3AttributeTypeNormal. The second field is a pointer to the array of attribute data (whether it be an array of vectors, floats, or whatever). This pointer is usually allocated dynamically; you'll seldom, if ever, have an array of 1,000 vectors (or whatever) waiting for you on the stack. This data pointer is (accordingly) something that you'll need to free up eventually, after you've successfully created your TriMesh. When QD3D gets your Q3TriMesh_New() call, it copies all of your attribute (and other) data into private, system-heap storage. After that, you no longer need to have big arrays of attribute data sitting around hogging your application's heap, so get rid of them — but not until you've called Q3TriMesh_New().

The final field of the TQ3TriMeshAttributeData, namely the attributeUseArray field, is only used in the case of custom (user-defined) attributes; most of the time you'll set this to nil.

Every type of attribute associated with any part of a TriMesh has to be specified in a TQ3TriMeshAttributeData structure. Of course, in QD3D, objects can have more than one type of attribute. In our case, we will want to be able to assign vertex normals, diffuse colors, and/or UV parameters (or all, or none of these) to our vertices. How can we do this? We do it with an *array* of TQ3TriMeshAttributeData structs — one for each attribute type.

So the game plan is: Fill out one TQ3TriMeshAttributeData struct for each type of attribute you need to put in the TriMesh, allocating data storage dynamically as needed. If you have more than one attribute type, make an *array* of TQ3TriMeshAttributeData structs. Then set the TriMesh's vertexAttributeTypes field to point to the *array* of TQ3TriMeshAttributeData. Got it?

I told you, the TriMesh takes getting used to.

Listing 3 shows not only how to stuff the right values into the right arrays, but also how to iterate through all of an object's attributes. Remember, in QD3D an object can have up to a dozen different attributes. In our case, we're only worried about three of the possible dozen types, but we may have to iterate a dozen times to find the three types we're interested in.

The call Q3AttributeSet_Get fetches the actual data we need (whether it's vector data, RGB data, or whatever) from the attribute set in question. Unlike Q3Geometry_GetAttributeSet, Q3AttributeSet_Get doesn't actually increase the reference count of anything, so there's no need to call Q3Object_Dispose afterward. In fact, if you do you'll run into weird errors.

Listing 3: ConvertGridAttributes()

ConvertGridAttributes()

```
long ConvertGridAttributes( TQ3TriGridData *tgData,
                            TQ3TriMeshData *tm)
{
  static TQ3TriMeshAttributeData attribs[3];
  TQ3AttributeType theType;
  long i,k;

    // We need to iterate thru all attribute TYPES and copy those we need
    // (namely UV params and/or vertex normals) into dynamically allocated arrays.

  for (i=0,
    theType = kQ3AttributeTypeNone,
    attribs[0].attributeType = nil,
    Q3AttributeSet_GetNextAttributeType(tgData-
                                >vertices[0].attributeSet,
        &theType );

    theType != kQ3AttributeTypeNone && i < 3; // loop termination

    Q3AttributeSet_GetNextAttributeType(tgData-
                          >vertices[0].attributeSet, &theType
))
  {

    attribs[i].attributeUseArray = nil;
    attribs[i].attributeType = theType;

    switch( theType ) {

      case kQ3AttributeTypeNormal :
      {
        TQ3Vector3D *vecs =
          (TQ3Vector3D *)NewPtr(sizeof(TQ3Vector3D)
                                          * tm->numPoints);

        for (k = 0; k < tm->numPoints; k++)
        {
      Q3AttributeSet_Get( tgData->vertices[k].attributeSet,
              kQ3AttributeTypeNormal,
              &vecs[k]);
        }

        attribs[i].data = vecs;
        i++;
      }
      break;

      case kQ3AttributeTypeShadingUV :
      {
        TQ3Param2D *params =
          (TQ3Param2D *)NewPtr(sizeof(TQ3Param2D)
                                          * tm->numPoints);
        for (k = 0; k < tm->numPoints; k++)
        {
          Q3AttributeSet_Get( tgData-
                                >vertices[k].attributeSet,
              kQ3AttributeTypeShadingUV,
              &params[k]);
        }

        attribs[i].data = params;
        i++;
      }
      break;

      case kQ3AttributeTypeDiffuseColor :
      {
        TQ3ColorRGB *color =
          (TQ3ColorRGB *)NewPtr(sizeof(TQ3ColorRGB)
                                          * tm->numPoints);
        for (k = 0; k < tm->numPoints; k++)
        {
          Q3AttributeSet_Get( tgData-
                                >vertices[k].attributeSet,
              kQ3AttributeTypeDiffuseColor,
              &color[k]);
        }

        attribs[i].data = color;
        i++;
      }
      break;
      default:
      break;

    } // switch

  } // end for loop

  tm->vertexAttributeTypes = attribs;
// now point at the TQ3TriMeshAttributeData array

  return i; // return the number of attributes copied
}
```

Now at last we're ready to tackle the actual creation of a TriMesh, which is the subject of **Listing 4**. Our custom data structure (pointed to by theDocument), contains a model that already has our TriGrid geometry loaded in it. So the first order of business is to try to fetch the TriGrid from the model. If we succeed, the next order of business is to start filling out the fields in our TriMesh's data structure. Note that many of the fields are set to zero or nil.

In **Listings 1, 2,** and **3** we showed how to copy the point, triangle, and attribute data from our TriGrid into the TriMesh's data structure. (Hence the calls to ConvertVertices, ConvertGridTriangles, and ConvertGridAttributes.) Finally, we call our own routine, GetTriMeshBBox, which we haven't shown here (but is provided in the complete project online); it simply finds the minimum and maximum x, y, and z values for the furthest-apart points in the grid. Then it's time to call Q3TriMesh_New. Generally speaking, if you've messed up anything prior to now, Q3TriMesh_New will return nil. (QD3D does a lot of internal consistency checks to spot potential problems before instantiating a TriMesh. If your data fields are bad, QD3D will refuse to create the mesh.)

The rest of the code in **Listing 4** is cleanup code to free arrays that have been dynamically allocated and get rid of our TriGrid reference as well as the data we retrieved from it. If you fail to make these calls, you can expect memory leaks.

Listing 4: DoTriMeshConversion()

DoTriMeshConversion()

Note: Error checking has largely been eliminated in the interest of clarity.

```
void DoTriMeshConversion( DocumentPtr theDocument ) {

  TQ3TriGridData      tgData;
  TQ3TriMeshData      trimesh;
  TQ3GeometryObject   tri;
  TQ3Status           s;
  TQ3GroupPosition    pos;
  long                i;

      // Get the first trigrid position from our group...
  s = Q3Group_GetFirstPositionOfType( theDocument->fModel,
      kQ3GeometryTypeTriGrid, &pos );
  if (s != kQ3Success) return;

      // now get the trigrid
  s = Q3Group_GetPositionObject( theDocument->fModel,
                                          pos, &tri);

  if (Q3TriGrid_GetData( tri, &tgData ) != kQ3Success)
    return;

  trimesh.triMeshAttributeSet = nil;
  trimesh.numPoints = tgData.numColumns * tgData.numRows;
  trimesh.numEdges= 0;
  trimesh.edges   = nil;
```

```
    trimesh.numTriangleAttributeTypes  = 0;
    trimesh.triangleAttributeTypes     = nil;
    trimesh.numEdgeAttributeTypes      = 0;
    trimesh.edgeAttributeTypes         = nil;

        // transcribe our geometry...
    ConvertVertices ( &tgData, &trimesh );
    ConvertGridTriangles( &tgData,
&trimesh,&trimesh.numTriangles);

        // transcribe our attributes...
    trimesh.numVertexAttributeTypes =
      ConvertGridAttributes(&tgData, &trimesh);

    GetTriMeshBBox( &trimesh ); // calculate bounding box

        // now create the TriMesh
    if (trimesh.triangles != nil && trimesh.points != nil)
      theDocument->fAlternateGeometry = Q3TriMesh_New( &trimesh
);

        //********* CLEANUPS *********
    if (trimesh.triangles != nil)
      DisposePtr((Ptr)trimesh.triangles);

    if (trimesh.points != nil)
      DisposePtr((Ptr)trimesh.points);

    for (i = 0; i < trimesh.numVertexAttributeTypes; i++)
      if (trimesh.vertexAttributeTypes[i].data != nil)
        DisposePtr((Ptr)trimesh.vertexAttributeTypes[i].data );

    Q3TriGrid_EmptyData( &tgData );    // free the trigrid data
    Q3Object_Dispose(tri);             // get rid of trigrid object reference
}
```

Does It Work?

Converting our terrain to a TriMesh speeds things up considerably. To benchmark the performance, I wrote a short routine, DoRenderTest, that rotates our object 360 degrees around the y-axis, in 30 increments of 12 degrees, and times the results in ticks (sixtieths of a second). On my 250Mhz G3 machine, a 4,096-vertex test object (with 7,938 polygons) takes 518 ticks to do a complete rotation as a TriGrid, but only 282 ticks to do it as a TriMesh. This is for an unsmoothed, untextured grid.

With the original 512-by-512-pixel source image applied as a texture map, again unsmoothed, the test cycle takes 773 ticks to complete in TriGrid mode but only 438 ticks for the TriMesh.

The performance comparison is even more interesting when vertex normals are included as attributes for smoothing (Gouraud shading). The textured, smoothed object takes 690 ticks to rotate as a TriGrid, but only 261 ticks as a TriMesh — a better than 2.5-to-1 speedup. Notice that adding vertex normals shaves almost 100 ticks off the TriGrid's time and *more than* 100 ticks off the TriMesh's time. This is because when we provide the renderer with precalculated, pre-cached vertex normals, the renderer doesn't have to calculate any surface normals on the fly.

Lesson No. 1: Always supply vertex normals, whenever you can. The normals don't have to be Gouraud-averaged, if you don't want smoothing. Just be sure to provide normals of *some* kind. (Code for doing this is in our project.)

Lesson No. 2: Use a TriMesh as your main "renderable" geometry, if possible.

If you apply *both* lessons, you should be able to see better than two-to-one speed gains on many (if not most) kinds of objects.

Less Is More

A mentor once gave me some truly excellent advice on how to get code to run faster. "The CPU can only execute so many instructions per second," he noted. "So in a sense, there is no such thing as *making code go faster.* There is only such a thing as making the machine *do less.*" I've often wanted to engrave those words in bronze. *To go fast, do less.* Let that thought percolate through your brain whenever you try to "speed up" your code.

It might behoove us to try to make our little **PMB** app do less. In this respect, you may have noticed that **PMB** isn't terribly smart about how it allocates resources. The flat parts of our terrain, for example, get just as many vertices as the "peaky" parts, even though — clearly — the flat parts could get by on less geometry. Fixing this ought to be easy: just delete triangles in flat areas. But wait: This isn't possible with the TriGrid geometry, which relies on fixed rows and columns.

Once again, TriMesh to the rescue.

By adding a few lines of code to our TriGrid-to-TriMesh triangle translation routine (to check the "altitude" of each point before copying), we can ensure that "sea-level" triangles don't get copied over to the TriMesh. With another line or two of code, we can add a "Trim Excess" toggle to our Preferences menu. The result is that wherever our terrain has zero-elevation triangles, the TriMesh version can be made to omit those triangles (and the associated point and attribute data) altogether, which ought to speed up rendering.

Figure 2. *Test image (scanned penny).*

Our test image (**Figure 2**) is a 512-by-512-pixel color PICT made by scanning a penny on a Microtek ScanMaker II-XE. You'll notice that the edges of the source image are white, which means lots of "edge" triangles can be eliminated. Sure enough, when we use the "Trim Excess" option in our app (see **Figures 3** and **4**), we find it's possible to eliminate over 3,000 polygons (out of 7,938) — which, of course, gives a welcome speed boost. In the rotation test, the spin time drops, in one instance, from 636 ticks for the TriGrid version to 182 for the trimmed TriMesh. (That's with a smoothed, textured object.) The untrimmed TriMesh turns in times in the 260-ticks area, so the speedup is apparently comparable to the reduction in polygon count (as you'd expect).

Figure 3. *Penny as terrain.*

There is much more to do in the "geometry optimization" department, if you think about it. After all, not only are triangles in *flat* spots (areas of zero slope) superfluous, but triangles in large areas of *constant* slope are redundant as well. With not much work, one could write a routine that loops over triangles in the mesh, examining all of a triangle's neighbors for coplanarity, the idea being that adjacent, coplanar polygons should be merged. Of course, this only works if the adjoining triangles are, in fact, mergeable: i.e., *right* triangles joined on a "small" edge (not the hypoteneuse). If you've done a "lazy rasterization" of triangles into the TriMesh — as I've done in our project (to keep the code small) — then *none* of the triangles will be mergeable, because they're all oriented the same way. (Merging any two of them will give a parallelogram.) Awhile ago, I mentioned that if you were creative, you could alter the hypoteneuse orientation of triangles as you copied them into the TriMesh. Now you know why. (Even so, not all triangle combos will be eligible for merging. Prove that, at most, two thirds will be.)

Figure 4. *Trimmed geometry.*

To reduce the triangle count of our grid to some kind of theoretical minimum requires that we go beyond mere *polygon decimation* and start over, using an *adaptive sampling technique* — that is, a technique that puts more vertices in areas of rapid slope change and fewer vertices in areas of less-rapid slope change. One possibility here is to develop an "area operator" (2D convolution kernel) that is sensitive to areas of high *standard deviation* of pixel luminance in the original image. Or you could simply look at the difference in pixel intensity between a central pixel and its north, south, east, and west neighbors. If north minus "center" equals "center" minus south, and east minus "center" equals "center" minus west, then by definition the pixel is in an area of zero slope change and needn't be made into a grid vertex.

Using an adaptive sampling technique of one sort of another will give you a point list containing vertices heavily concentrated in areas of rapid slope change and less concentrated in constant-slope areas. The next trick is to convert this vertex assortment into triangles. Triangulation of an arbitrary point list is a classical problem in 3D geometry and is harder, frankly, than it first seems. One difficulty you run into if you start connecting points indiscriminately is that you quickly end up with lots of long, skinny (scalene) triangles that don't really represent the underlying topography very well. (A "raster-line" approach gives notably poor results.) If you're interested in pursuing this subject further, search on the World Wide Web using "Delaunay" as a keyword. Also, be sure to consult the various *Graphics Gems* volumes published by AP Professional. (Look in the index under "triangulation" and "tesselation.")

Tweaks

From this point on, we're looking at relatively minor "tweaks" to improve performance. One that deserves mention involves monitor depth and pixel size. As simple as it sounds, a 32-bit texture uses twice the memory of a 16-bit texture, so when you're using texture maps you should draw them into 16-bit GWorlds and create 16-bit storage pixmaps. You should also set your monitor to 16-bit color mode if you want to see things render quickly. In general, when you can cut the video-byte traffic by half, you'll see a certain amount of performance improvement, even on a machine with special video hardware. The performance improvement may not be much, but it will be there.

In writing this article, I had a chance to experiment with changing all of the texture-map code to reflect 16-bit pixels (versus 32-bit), and in testing I saw a 5% speed improvement with 16-bit-textured objects running on a monitor set to "thousands of colors," versus the same objects with 32-bit textures and 32-bit monitor depth. Admittedly, this is not a terribly important performance gain. A better reason to use 16-bit textures is that you won't run out of VRAM so quickly when using big textures.

Incidentally, almost all 3D accelerator cards require that texture maps have pixel dimensions that are a power of 2 (such as 128, 256, 512, etc.), and in some cases you'll get a performance increase if you stick with *square* images (128-by-128 instead of, say, 128-by-256). It's also a good idea to render into windows that are aligned on a 32-byte boundary (the size of a PowerPC cache line) and try for window dimensions that are multiples of 32.

Homegrown Math Routines

If you've got a 3D application or subroutine that does a *lot* of matrix or vector math, you'll want to consider hand-writing your own math functions rather than using QD3D's extensive built-in math library. In the bad old days of CISC chips with very little cache, no floating-point unit, etc., you had little choice but to roll your own math routines. Today, it's not such a pressing necessity. With the advent of the PowerPC architecture (with its big instruction cache, onboard floating-point unit, and abundance of registers), function calls don't *have* to mean lots of overhead — most compilers nowadays pass arguments in registers rather than on the stack — but just the same, you'll probably find that it pays to inline some math routines of your own now and then, particularly in tight loops that handle thousands of vertices or polygons at a time. It's doubtful you'll actually want to drop down into assembly language,

for a variety of reasons. But you may find that using your own matrix-math routines can spare QD3D from having to reload matrices over and over inside a loop, for example. Some of the QD3D math routines have error-checking overhead that (if you're careful) you can profitably sidestep by using your own routines.

One trick I've often found useful is to eliminate square roots whenever possible. Sometimes you're just comparing the lengths of two line segments, for example, in which case you *don't need* to take the square root of the sum of the squares. Just compare the squares. You'll get the same result. (A longer distance, squared, is always longer than a shorter distance, squared.)

Other QD3D Tricks

There's a little-known QD3D call that (in theory) can affect the speed with which textured objects are drawn. It's called Q3InteractiveRender_SetRAVETextureFilter(). There are 3 predefined constants that can be passed in this call. If you look in Rave.h you'll find them:

```
#define kQATextureFilter_Fast   0
#define kQATextureFilter_Mid  1
#define kQATextureFilter_Best 2
```

The idea here is that you can control the degree to which the renderer tries to subpixel-filter your textures when objects are close to the camera. If you don't mind a "chunky" look (i.e., you'd rather have faster rendering), you can select kQATextureFilter_Fast and have faster rendering at the expense of not-as-pretty texturing. According to Apple's QD3D development team, this scheme isn't actually implemented in QD3D 1.5.4, but will be in a future release. So be ready for it.

If you have a "busy" scene with lots of objects or you do a lot of camera fly-throughs, you should probably consider doing your own object culling. *Culling* refers to the process of determining which objects, if any, in a scene are outside the viewing range of the camera and therefore needn't be submitted to the renderer. QD3D's interactive renderer does its own culling, but not very efficiently. Your application (unlike the renderer) may "know" a lot about the scene, *a priori*, and as a result you can often reduce the number of objects submitted for rendering. (Remember: To go fast, do less.)

For more ideas on how to speed up QD3D, be sure and track down the excellent document, "Making Cool QuickDraw 3D Applications" by Apple's Brian Greenstone. This document has changed locations a lot but is on Apple's web site. (Search on "cool3dappspdf.") It contains lots of tips for making QD3D objects render faster.

Conclusion

QuickDraw 3D lives up to its name in most situations, but in QD3D programming (as in all graphics work) there's no such thing as too much speed. In our sample application, we saw that by means of a few relatively simple changes to our code (if you can call translating a TriGrid to a TriMesh simple) we were able to score around a 5-to-1 performance gain, taking a couple-frames-per-second update rate to well over 10 fps on a 250Mhz machine. (In its final version, our test object completed its 30-frame test cycle in less than 150 ticks — versus more than 750 ticks for the worst-case scenario.)

Some of the lessons we learned were:

- Always use the TriMesh data structure where speed is the main concern. This is generally good for at least a two-to-one speedup.
- Reduce the geometry to the absolute minimum number of points and polygons needed to get the job done. We found that simply trimming the edges of our geometry eliminated large numbers of unneeded vertices, giving a corresponding speed boost.
- Apply vertex-normal attributes whenever you can. Supplying precalculated vertex normals keeps the renderer from having to calculate its own normals on the fly.
- Use 16-bit texture maps and 16-bit monitor mode to reduce VRAM traffic.
- Size textures to a power of two on each dimension.
- Size windows to a multiple of 32 pixels in each dimension.
- If any "tight loops" use vector or matrix math routines, write your own routines inline rather than calling out to the QD3D math library.
- In a multi-object scene where some objects are bound to be out of viewing range, do your own culling. Don't just submit all objects to the renderer. The renderer's culling routine is seldom going to be as fast as one you come up with.
- Most of all: To go fast, do less. The biggest speed gains of all sometimes come when you can figure out a way not to have to calculate something at all.

MT

PROGRAMMER'S CHALLENGE

by Bob Boonstra, Westford, MA

Word Neighbors

Several of you have written me to point out that the Challenges have been getting more and more difficult over the years. Thinking back over the past few months, where we have solved an orbital mechanics problem, programmed a winning Hearts solution, and written an emulator for the first stored-program computer, I can see where this might be true. This month the problem is easier. Your Challenge is to write a search routine that will find a set of words that are near to one another in some target text. You might imagine the solution being used as part of an internet search engine, providing a capability more selective than looking for all of the target words being present on a page, but less restrictive than requiring the target words to form a phrase.

The prototype for the code you should write is:

```
#if defined(__cplusplus)
extern "C" {
#endif

pascal void Initialize(
   char *text,                /* NULL terminated text to be searched */
   long distance,             /* max distance between nearby words */
   void *privateStorage,      /* private storage for your use */
   long storageBytes          /* number of bytes in privateStorage */
);

pascal long FindNearby(       /* return number of matches found */
   char *words[],             /* words to find in text */
   long numWords,             /* number of words */
   Boolean caseSensitive,     /* true if match is case sensitive */
   Boolean preserveOrder,     /* true if words must be found in order */
   long matchPositions[],     /* position in text of first word in match */
   long maxMatches            /* max number of matches to return */
);

#if defined(__cplusplus)
}
#endif
```

Your Initialize routine is called to give you an opportunity to preprocess the text to be searched. The text consists of a sequence of words separated by delimiters. For the purpose of this Challenge, a word is a sequence of alphanumeric characters separated by one or more non-alphanumeric characters. The text may include ASCII characters between 0x20 and 0x7E, inclusive, plus carriage returns (0x0D), linefeeds (0x0A), and tabs (0x09). All non-alphanumeric characters are delimiters.

You are given the maximum distance to be allowed between adjacent search words. A distance of 0 corresponds to adjacent words, a distance of 1 corresponds to search words with one intervening word, etc. You are also given a privateStorage pointer to storage available for your use, along with the size (storageBytes) of privateStorage in bytes. There will be at least 16 bytes of storage for each unique (case-sensitive) word in the text, plus 4 bytes for each word occurrence.

For each time that Initialize is called with new text to be searched, your FindNearby routine will be called an average of 10 to 50 times with a set of numWords words to find. You will be told whether the search is to be caseSensitive or not, and whether the words must be found in order (preserveOrder==TRUE) in the text. You must find the first maxMatches occurrences of the words in the text, where a match occurs when each of the words is separated by no more than a maximum number (distance) of intervening words from another of the search words. For each match, you should return in matchPositions the offset in text of the first of the search words found in text. For example, if distance==2 and the text is "The quick brown fox jumps over the lazy dog.", you would return 4 if the words were "brown", "quick", and "jumps", provided preserveOrder is FALSE. No word in the text may be part of more than one match.

The winner will be the solution that correctly finds the word matchPositions in the minimum execution time, including the time taken by the Initialize routine..

This will be a native PowerPC Challenge, using the latest CodeWarrior environment. Solutions may be coded in C, C++, or Pascal.

Three Months Ago Winner

Congratulations once again to **Ernst Munter (Kanata, Ontario)** for submitting the fastest solution to the September "Big Baby" Challenge. Eleven people submitted entries that simulated the operation of the Manchester Mark 1 prototype computer, the world's first stored program computer, which was fifty years old this past June. Ernst was one of four entries to take advantage of our annual September opportunity to include assembly language in their solution.

One of the novel things about the winning solution is the way Ernst executes instructions in pairs, trying to take full advantage of the PowerPC processor. The solution accurately models an unusual overflow characteristic of the original Baby computer, described in the Baby documentation and mentioned in the solution comments. However, because this overflow behavior occurs infrequently, the winning solution detects an overflow in its fast mode of execution and switches to a slower mode of operation to process it.

The problem statement required contestants to treat memory the way it appeared on the Baby CRT display, with the least significant bit on the left. Some of the solutions did this via enormous table lookups, leading to a large memory

footprint. Ernst did the bit-flipping in his CopyMem routine using the TWIDDLE macro. He uses one of the PowerPC's most versatile instructions, the Rotate Left Word Immediate Then And With Mask, or rlwinm instruction, to first flip bytes, then 4-bit nibbles, then 2-bit chunks, and finally individual bits.

I used nine test cases to evaluate the Big Baby solutions, including four that executed the Assembler function. While the original Baby computer used only 5 address bits, allowing it to address 32 words of memory, the Big Baby Challenge required Challengers to deal with larger memory spaces, and my test cases used up to 8 address bits. The test cases included two programs demonstrating self-modifying code: a 16-bit counter program submitted by Ernst Munter, and a BabyFill program submitted by Peter Lewis that counts the number of words of memory available. (Both Ernst and Peter earn two points for submitting these programs.) The test cases also included two programs submitted to the University of Manchester Baby contest last June, a prime number generation program, and a "3 minute noodle timer" program that is quite interesting to watch execute on the graphical Baby simulator available at the University of Manchester web site. The 16-bit counter, BabyFill, and prime number programs are reproduced below.

16-bit counter program

```
32
0000 JMP 18
0001 LDN 20
0002 JRP 19
0003 NUM 536870911
0004 NUM 134217727
0005 NUM 33554431
0006 NUM 8388607
0007 NUM 2097151
0008 NUM 524287
0009 NUM 131071
0010 NUM 32767
0011 NUM 8191
0012 NUM 2047
0013 NUM 511
0014 NUM 127
0015 NUM 31
0016 NUM 7
0017 NUM 1
0018 JMP 21
0019 NUM 23
0020 NUM -24593
0021 JRP 0
0022 LDN 30
0023 STO 30
0024 LDN 30
0025 SUB 17
0026 STO 30
0027 SUB 21
0028 STO 29
0029 NUM 0
0030 NUM 0
0031 CMP
```

```
BabyFill
16
0000 NUM 0
0001 JRP 4
0002 STO 0
0003 NUM -1
0004 NUM 6
0005 NUM 1
0006 NUM -32771
0007 NUM -14
0008 LDN 14
0009 SUB 5
0010 STO 14
0011 LDN 14
0012 STO 14
0013 LDN 6
0014 STO 15
0015 LDN 7
```

```
Prime Number generator
30
0000 JMP 24
0001 LDN 21
0002 STO 21
0003 LDN 21
0004 SUB 15
0005 STO 21
0006 LDN 15
0007 STO 22
0008 LDN 22
0009 STO 22
0010 LDN 22
0011 SUB 15
0012 STO 22
0013 SUB 21
0014 CMP
0015 NUM -1
0016 LDN 21
0017 STO 23
0018 LDN 23
0019 SUB 22
0020 JMP 0
0021 NUM 1
0024 NUM 7
0025 CMP
0026 JRP 0
0027 STO 23
0028 LDN 23
0029 SUB 22
0030 CMP
0031 JMP 20
```

The table below lists, for each of the solutions submitted, the total execution time, the number of the nine test cases that the entry processed incorrectly, and the execution times of three individual test cases. Test case 3 is the 16-bit counter mentioned above. Case 6 is the prime number generator, which was executed 30 times for each entry, producing the 30th prime number as the final result. Test case 8 was the "3 minute" noodle timer, which executed in 3 minutes on the original Baby, but which would leave our noodles seriously undercooked on my 200MHz 604e. You can see that the solutions rank in the same speed order for each of the individual test cases as they do in the aggregate. The table also includes code size, data size, and programming language used for each solution. As usual, the number in parentheses after the entrant's name is the total number of Challenge points earned in all Challenges prior to this one. Four of the 11 entries crashed or hung my machine; I've replaced the author's name with their initials in those cases.

Name/Lang	Time	Errors	Case3	Case6	Case8	Code	Data
Ernst Munter (408) C++/asm	41.01	0	29.67	2.99	6.56	9636	104
ACC Murphy (34) asm	54.46	0	39.37	4.20	8.20	2640	16
Rob Shearer C++	64.10	0	46.54	5.63	8.91	2340	131246
Rob Managan C	77.85	0	55.25	7.24	11.66	1940	324
Daniel Harding C++	110.10	0	78.90	9.49	15.98	1424	8632
Randy Boring (83) C/asm	69.65	1	49.87	5.38	10.20	1356	1120
Willeke Rieken (47) C	70.08	2	51.04	5.93	10.49	1748	16
A. C. C		Crash				892	692
M. P. asm		Crash				640	131104
T. B. C		Crash				2248	116
D. F. C		Hang				1056	237

Top Contestants

Listed here are the Top Contestants for the Programmer's Challenge, including everyone who has accumulated 20 or more points during the past two years. The numbers below include points awarded over the 24 most recent contests, including points earned by this month's entrants.

Rank	Name	Points	Rank	Name	Points
1.	Munter, Ernst	206	10.	Nicolle, Ludovic	34
2.	Boring, Randy	56	11.	Lewis, Peter	31
3.	Mallett, Jeff	50	12.	Hart, Alan	21
4.	Saxton, Tom	49	13.	Antoniewicz, Andy	20
5.	Rieken, Willeke	47	14.	Day, Mark	20
6.	Cooper, Greg	44	15.	Higgins, Charles	20
7.	Maurer, Sebastian	40	16.	Hostetter, Mat	20
8.	Heithcock, JG	37	17.	Studer, Thomas	20
9.	Murphy, ACC	34			

There are three ways to earn points: (1) scoring in the top 5 of any Challenge, (2) being the first person to find a bug in a published winning solution or, (3) being the first person to suggest a Challenge that I use. The points you can win are:

1st place	20 points
2nd place	10 points
3rd place	7 points
4th place	4 points
5th place	2 points
finding bug	2 points
suggesting Challenge	2 points

Here is Ernst's winning Big Baby solution:

Baby.cp


```
/*

Version 3

The Problem
-----------
Write an assembler and emulator for "Baby" Mark I, the first
computer to store a program electronically (back in 1948).
Instead of just 32 words of store, "BigBaby" can have up to
1024 words.

The emulator should handle all programs in the same way as the
MISIM simulator of the University of Manchester. This includes
the peculiar overflow behaviour associated with the CI / PI
interaction.

Bit flipping
------------
The Baby convention of displaying the LSB as bit 0 on the left
is also emulated, in effect by reversing the bits in memory
before and after execution. This feature can be turned off.

The Assembler
-------------
The assembler is unremarkable. It happens that the opcodes for
this small set of instruction mnemonics can be separated by a
simple hash function, and then translated to the actual codes
through a very small table. This is quicker than a switch
statement or a series of compare/branch.

The Emulator
------------
The emulator is "lazy" about the peculiar CI / PI behaviour
which is not expected to come up with most programs. Only
enough logic is provided in the fast part of the emulator to be
able to detect CI overflow as an "exception". When such an
exception occurs, the emulator switches over into a slower but
accurate emulation of the particular Baby design, and the
program then finishes in that part.

Both parts of the emulator are structured around computed jumps
to a set of instruction blocks which emulate each Baby opcode.

Slower Loop
-----------
The slower section selects one instruction at a time,
and processes it to completion before fetching the next
instruction.

Faster Loop
-----------
The faster loop of the emulator attempts to execute two instructions
in parallel, thus striving to exploit the multiple integer
execution units in the powerPC chip.

This results in a 64-way switch, for all combinations in pairs
of Baby opcodes. In addition, some instruction pairs lend
themselves to interleaving of the execution of the present two
instructions with fetching and decoding of the next two.

In a previous version, I spread the fetch, decode, and execute
of three instructions over three consecutive blocks, thus
having effectively 3 instructions in the pipeline. The new
version is slightly faster.

Wrapping
--------
When the end of the defined memory is reached, the program must
start again at location 0. In the slower loop (the "standard"
implementation of the algorithm), program addresses are derived
from CI by ANDing with a mask, and wrapping is automatic.

In the faster loop, I use a self-incrementing program counter
which directly addresses memory. This incurs some overhead when
the end of the defined memory is reached. I have extended the
```

memory (a copy) by two sentinel locations which are filled with STP instructions. When a STP is reached, a quick comparison determines if this is the true stop, or a sentinel. In the latter case, program execution is restarted at the first or second location.

PI Addition Mode

Under certain conditions, a program instruction is created by the addition of the previous instruction with the present one. This condition (bit 13 set in CI) can only arise at three points:
- a JMP (indirect JUMP)
- a JRP (relative JUMP)
- when wrapping, as CI counts up.

All three occasions are already separated in the program, and allow a simple check of the state of CI. When the exceptional case is detected, program flow is directed into the slower loop which deals with the instruction addition as well as the possible "fatal" error, when CI-bits 14 or 15 are set. In that case, the emulator program simply returns.

A note on the M1SIM implementation of this:
CI is incremented at the start of each instruction, but the check of CI-bit 13 is made before incrementing CI. In this way, it is possible for CI to emerge with bit 13 set, but without instruction addition having taken place (but it likely will happen in the next cycle).

Assumptions

The constant kMaxInstructions is only used to allocate memory, the memory size used by the Baby emulator algorithm is always memUsed = (1 << address_bits), i.e. all memory address calculations are modulo memUsed.

If kMaxInstructions == (1 << address_bits), many PPC instructions can be saved by combining shift and mask in a single opcode (rlwinm) which expects constant parameters. To exploit this, I have provided two complete copies of the Execute(..) function where the correct one is automatically selected. The gain in speed is about 7% when kMaxInstructions matches address_bits.

```
*/

# include "Baby.h"

/*
 Two versions of the memory copy function are provided.
 One which reverses the bit order of each 32-bit word,
 the other just a normal copy.

 The pre-processor directive UNUSUAL_BIT_ORDER selects.
*/

# define BIT_ORDER_REVERSED 1

#if BIT_ORDER_REVERSED

static UInt32 RM[4]={
  0xFF00FF00,0xF0F0F0F0,0xCCCCCCCC,0xAAAAAAAA
};

// Memory copy while flipping bits
static asm void CopyMem(
  UInt32* dest,UInt32* src,UInt32 num) {
# define mask8 r7
# define mask4 r8
# define mask2 r9
# define mask1 r10
  lwz    r6,RM(RTOC)
  lwz    mask8,0(r6)
  lwz    mask4,4(r6)
  lwz    mask2,8(r6)
  lwz    mask1,12(r6)

  mtctr    r5
@loop:

  lwz    r0,0(r4)
  addi   r4,r4,4
  rlwinm r5,r0,16,0,15
  rlwinm r6,r0,16,16,31
  add    r0,r5,r6
```

```
# define TWIDDLE(x)          \
  rlwinm r5,r0,x,0,31-x;    \
  and    r6,mask##x,r0;     \
  and    r5,mask##x,r5;     \
  rlwinm r6,r6,32-x,x,31;\
  add    r0,r5,r6;

  TWIDDLE(8)
  TWIDDLE(4)
  TWIDDLE(2)
  TWIDDLE(1)

  stw    r0,0(r3)
  addi   r3,r3,4
  bdnz   @loop
  blr
}

# else

// Simple memory copy
static asm void CopyMem(
  UInt32* dest,UInt32* src,UInt32 num) {
  rlwinm r0,r5,30,2,31
  mtctr    r0
@loop:
  lwz    r5,0(r4)
  lwz    r6,4(r4)
  lwz    r7,8(r4)
  lwz    r8,12(r4)
  addi   r4,r4,16
  stw    r5,0(r3)
  stw    r6,4(r3)
  stw    r7,8(r3)
  stw    r8,12(r3)
  addi   r3,r3,16
  bdnz   @loop
  blr
}

#endif

/*
 Assembler for the Baby code mnemonics, with supporting functions.
*/

enum{
  kOpShift = 13,
  JMP=0,JRP,LDN,STO,SUB,SU2,CMP,STP,NUM=0
};

const int xlateOP[16]={
  JRP,STO,-1,-1,-1,-1,-1,CMP,LDN,-1,NUM,-1,-1,SUB,STP,JMP
};

inline int GetUnsigned(char* & p) {
  int c,n=*p - 48;
  while ((c=*++p)>='0')
    n=n*10+c-48;
  return n;
}

inline int GetSigned(char* & p) {
  if (*p == '-') {
    return -GetUnsigned(++p);
  } else return GetUnsigned(p);
}

inline long GetOpcode(char* & p) {
  long OP=*((long*)(p));
  p+=4;
// The following hash function separates all legal
// op code mnemonics into some value between 0 and 15.
// The xlateOP table then translates this to the correct
//    value of each op code.
  return xlateOP[(OP ^ (OP>>8) ^ (OP>>13)) & 15];
}

pascal void AssembleBabyProgram(
  char *program,
  CRT_memory memory,
  UInt32 address_bits) {
// Reads and assembles strictly formatted ASM Baby programs
// It is robust against unreasonable values, but
// incorrectly formatted input will give wrong the output
  UInt32 memSize=1<<address_bits;
  if (memSize>kMaxInstructions)
    return;
  UInt32 addrMask=memSize-1;
// read number of lines to follow
  int numLines=GetUnsigned(program);
  for (int i=0;i<numLines;i++) {
// first item is a 4-digit decimal number
// instructions can be in any order,
// missing address locations are not touched
    int addr=addrMask & GetUnsigned(++program);
// followed by a space and a 3-character op-code mnemonic
    int opcode=GetOpcode(program);
  memory[addr]=(opcode << kOpShift);
    if (opcode<CMP)
// only op codes below CMP are followed by a number
      memory[addr]+=GetSigned(++program);
  }

#if BIT_ORDER_REVERSED
//flip bits in each word
  CopyMem(memory,memory,memSize);
#else
//no need to use copy
#endif
}

/*
 Two very similar versions of the Execute function:
 the only difference is that Execute0 takes advantage of
 the fact when 1<<address_bits == kMaxInstructions to
 merge the and and shift functions into a single rlwinm
 opcode which requires constant parameters.
*/

# define nop3 nop;nop;nop
# define nop4 nop3;nop
# define nop5 nop4;nop
# define nop6 nop5;nop
# define nop7 nop6;nop
# define nop8 nop7;nop
# define nop9 nop8;nop
# define nop10 nop9;nop
# define nop11 nop10;nop
# define nop12 nop11;nop
# define nop13 nop12;nop
# define nop14 nop13;nop
# define nop15 nop14;nop

// volatile registers must be named explicitely
# define TEMP   r0
# define MEM    r3
# define addMask  r4
# define PC     r5
# define addShft1 r7
# define addShft2 r8
# define endPC    r9

static asm void Execute(CRT_memory memory,UInt32 memUsed) {
// non-volatile registers are assigned by the compiler
register int* memSpan;
register int* ACC;
register int* BASE;
register int* CI;
register int* M1;
register int* M2;
register int* PI1;
register int* PI2;
register int* addr1;
register int* addr2;
register int* instruction;
register int* CIinstField;

  fralloc

  rlwinm memSpan,addMask,2,0,29
  add    endPC,MEM,memSpan
  addi   addMask,addMask,-1
```

```
    b       @theCode
@GetCodeBase:
    mflr    BASE
    li    ACC,0
    li    CI,1

@restart:
    and     TEMP,CI,addMask
    addi    CI,CI,1
    rlwinm PC,TEMP,2,0,29
    add     PC,MEM,PC
    lwz     PI1,0(PC)
    lwzu    PI2,4(PC)
    b       @decode

@loop:
    lwz     PI1,4(PC)
    addi    CI,CI,2
    lwzu    PI2,8(PC)
@decode:
    rlwinm instruction,PI1,28,20,22
    rlwimi instruction,PI2,25,23,25
    and     addr1,PI1,addMask
    add     instruction,BASE,instruction
    mtctr     instruction
    and     addr2,PI2,addMask
    rlwinm addShft1,addr1,2,0,29
    rlwinm addShft2,addr2,2,0,29
// computed jump to the selected pair of instructions
    bctr

@theCode:
    bl    @GetCodeBase

#define doJMP1                          \
    lwzx      CI,MEM,addShft1;      \
    addi    CI,CI,1;                    \
    rlwinm.   CIinstField,CI,0,16,18;   \
    beq+    @restart;                   \
    mr    PI1,PI2;                      \
    b       @exception

#define doJMP2                          \
    lwzx      CI,MEM,addShft2;      \
    addi    CI,CI,1;                    \
    rlwinm.   CIinstField,CI,0,16,18;   \
    beq+    @restart;                   \
    b       @exception
#define doJRP1                          \
    lwzx      M1,MEM,addShft1;      \
    add     CI,M1,CI;                   \
    rlwinm.   CIinstField,CI,0,16,18;   \
    beq+    @restart;                   \
    mr    PI1,PI2;                      \
    b       @exception
#define doJRP2                          \
    lwzx      M2,MEM,addShft2;      \
    addi    CI,CI,1;                    \
    add     CI,M2,CI;                   \
    rlwinm.   CIinstField,CI,0,16,18;   \
    beq+    @restart;                   \
    b       @exception
#define doLDN(x)                        \
    lwzx      M2,MEM,addShft##x;    \
    neg     ACC,M2
#define doLDN_STO                       \
    lwz     PI1,4(PC);                  \
    lwzu    PI2,8(PC);                \
    lwzx      M1,MEM,addShft1;      \
    addi    CI,CI,2;                    \
    rlwinm instruction,PI1,28,20,22;\
    rlwimi instruction,PI2,25,23,25;\
    and     addr1,PI1,addMask;        \
    add     instruction,BASE,instruction;\
    mtctr     instruction;              \
    neg     ACC,M1;                     \
    and     addr2,PI2,addMask;        \
    rlwinm addShft1,addr1,2,0,29; \
    stwx      ACC,MEM,addShft2;       \
    rlwinm addShft2,addr2,2,0,29; \
    bctr
```

```
#define doLDN_SUB                          \
  lwz    PI1,4(PC);                        \
  lwzu   PI2,8(PC);                        \
  lwzx     M1,MEM,addShft1;      \
  addi   CI,CI,2;                          \
  rlwinm instruction,PI1,28,20,22;\
  rlwimi instruction,PI2,25,23,25;\
  lwzx   M2,MEM,addShft2;        \
  and    addr1,PI1,addMask;      \
  add    instruction,BASE,instruction;\
  mtctr    instruction;                    \
  neg    ACC,M1;                           \
  and    addr2,PI2,addMask;      \
  rlwinm addShft1,addr1,2,0,29; \
  subf   ACC,M2,ACC;                       \
  rlwinm addShft2,addr2,2,0,29; \
  bctr
#define doSTO1(1)                          \
  add    TEMP,addShft1,MEM;      \
  cmpw     PC,TEMP;                        \
  stwx     ACC,MEM,addShft1;       \
  bne+   @##1;                                \
  lwz    PI1,0(PC);                        \
  lwzu   PI2,4(PC);                        \
  addi   CI,CI,1;                          \
  b    @decode;                            \
@##1
#define doSTO2                                \
  stwx     ACC,MEM,addShft2
#define doSUB(x)                           \
  lwzx     M2,MEM,addShft##x;    \
  subf     ACC,M2,ACC
#define doSUB_STO                          \
  lwz    PI1,4(PC);                        \
  lwzu   PI2,8(PC);                        \
  lwzx     M2,MEM,addShft1;      \
  addi   CI,CI,2;                          \
  rlwinm instruction,PI1,28,20,22;\
  rlwimi instruction,PI2,25,23,25;\
  and    addr1,PI1,addMask;      \
  add    instruction,BASE,instruction;\
  mtctr    instruction;                    \
  subf     ACC,M2,ACC;                     \
  and    addr2,PI2,addMask;      \
  rlwinm addShft1,addr1,2,0,29; \
  stwx     ACC,MEM,addShft2;       \
  rlwinm addShft2,addr2,2,0,29; \
  bctr
#define doCMP1                                \
  cmpwi  ACC,0;                            \
  blt    @loop
#define doCMP2                                \
  cmpwi    ACC,0;                          \
  bge    @loop;                            \
  addi     PC,PC,4;                        \
  addi     CI,CI,1
#define doSTP1                                \
  cmpw     PC,endPC;               \
  ble-     @exit;                          \
  addi   CI,CI,-1;                         \
  rlwinm.  CIinstField,CI,0,16,18;   \
  beq+   @restart;                         \
  mr   PI1,PI2;                            \
  b      @exception
#define doSTP2                                \
  cmpw     PC,endPC;               \
  blt-     @exit;                          \
  rlwinm.  CIinstField,CI,0,16,18;   \
  beq+   @restart;                         \
  b      @exception
// @AAA_BBB labels define the instruction pair, for readability
@JMP_JMP:doJMP1;nop10
@JMP_JRP:doJMP1;nop10
@JMP_LDN:doJMP1;nop10
@JMP_STO:doJMP1;nop10
@JMP_SUB:doJMP1;nop10
@JMP_SU2:doJMP1;nop10
@JMP_CMP:doJMP1;nop10
@JMP_STP:doJMP1;nop10

@JRP_JMP:doJRP1;nop10
@JRP_JRP:doJRP1;nop10
@JRP_LDN:doJRP1;nop10
@JRP_STO:doJRP1;nop10
@JRP_SUB:doJRP1;nop10
@JRP_SU2:doJRP1;nop10
@JRP_CMP:doJRP1;nop10
@JRP_STP:doJRP1;nop10

@LDN_JMP:doLDN(1);doJMP2;nop9
@LDN_JRP:doLDN(1);doJRP2;nop8
@LDN_LDN:doLDN(2);b @loop;nop13
@LDN_STO:doLDN_STO;nop
@LDN_SUB:doLDN_SUB
@LDN_SU2:doLDN_SUB
@LDN_CMP:doLDN(1);doCMP2;b @loop;nop9
@LDN_STP:doLDN(1);doSTP2;nop9

@STO_JMP:doSTO1(1);doJMP2;nop3
@STO_JRP:doSTO1(2);doJRP2;nop;nop
@STO_LDN:doSTO1(3);doLDN(2);b @loop;nop5
@STO_STO:doSTO1(4);doSTO2;b @loop;nop6
@STO_SUB:doSTO1(5);doSUB(2);b @loop;nop5
@STO_SU2:doSTO1(6);doSUB(2);b @loop;nop5
@STO_CMP:doSTO1(7);doCMP2;b @loop;nop3
@STO_STP:doSTO1(8);doSTP2;nop3

@SUB_JMP:doSUB(1);doJMP2;nop9
@SUB_JRP:doSUB(1);doJRP2;nop8
@SUB_LDN:doLDN(2);b @loop;nop13
@SUB_STO:doSUB_STO;nop
@SUB_SUB:doSUB(1);doSUB(2);b @loop;nop11
@SUB_SU2:doSUB(1);doSUB(2);b @loop;nop11
@SUB_CMP:doSUB(1);doCMP2;b @loop;nop9
@SUB_STP:doSUB(1);doSTP2;nop9

@SU2_JMP:doSUB(1);doJMP2;nop9
@SU2_JRP:doSUB(1);doJRP2;nop8
@SU2_LDN:doLDN(2);b @loop;nop13
@SU2_STO:doSUB_STO;nop
@SU2_SUB:doSUB(1);doSUB(2);b @loop;nop11
@SU2_SU2:doSUB(1);doSUB(2);b @loop;nop11
@SU2_CMP:doSUB(1);doCMP2;b @loop;nop9
@SU2_STP:doSUB(1);doSTP2;nop9

@CMP_JMP:doCMP1;doJMP2;nop9
@CMP_JRP:doCMP1;doJRP2;nop8
@CMP_LDN:doCMP1;doLDN(2);b @loop;nop11
@CMP_STO:doCMP1;doSTO2;b @loop;nop12
@CMP_SUB:doCMP1;doSUB(2);b @loop;nop11
@CMP_SU2:doCMP1;doSUB(2);b @loop;nop11
@CMP_CMP:b @loop;nop15  // a form of NOP !
@CMP_STP:doCMP1;doSTP2;nop9

@STP_JMP:doSTP1;nop9
@STP_JRP:doSTP1;nop9
@STP_LDN:doSTP1;nop9
@STP_STO:doSTP1;nop9
@STP_SUB:doSTP1;nop9
@STP_SU2:doSTP1;nop9
@STP_CMP:doSTP1;nop9
@STP_STP:doSTP1;

@exit:
  frfree
  blr

// If an exception (CI instruction bit set) was detected,
// execution continues here, with a loop which checks CI
// on every instruction.  Not much optimized since we don't
// expect to run in this mode very much
@exception:
//reconstruct CI:
  addi   CI,CI,-1
  b      @theCode2
@getCodeBase2:
  mflr   BASE
```

```
@L1:
// test CI bits 13-15 before incrementing CI
// this appears to be what M1SIM.EXE does
   rlwinm.   CIinstField,CI,0,16,18
   addi        CI,CI,1
   and     TEMP,CI,addMask
   rlwinm TEMP,TEMP,2,0,29
   lwzx        M1,MEM,TEMP    //load instr word from memory
   beq-        @L2
   cmpwi     CIinstField,0x2000
   bne-    @exit                    // return if CI instruction != 1
   add     PI1,PI1,M1          // add instruction to instruction !!
   b           @L3
@L2:
   mr          PI1,M1             // the normal case : PI = CW
@L3:
   rlwinm      instruction,PI1,23,25,27     // instruction * 16
   add     instruction,BASE,instruction
   mtctr       instruction
   and     addr1,PI1,addMask
   rlwinm addShft1,addr1,2,0,29
   bctr
@theCode2:
   bl   @getCodeBase2
@JMP:
   lwzx CI,MEM,addShft1
   b       @L1;       nop;nop
@JRP:
   lwzx    TEMP,MEM,addShft1
   add     CI,TEMP,CI
   b       @L1;       nop
@LDN:
   lwzx    TEMP,MEM,addShft1
   neg     ACC,TEMP
   b       @L1;          nop
@STO:
   stwx    ACC,MEM,addShft1
   b       @L1;       nop;nop
@SUB:
   lwzx    TEMP,MEM,addShft1
   subf    ACC,TEMP,ACC
   b       @L1;          nop
@SUB2:
   lwzx    TEMP,MEM,addShft1
   subf    ACC,TEMP,ACC
   b       @L1;          nop
@CMP:
   cmpwi ACC,0
   bge     @L1
   addi    CI,CI,1
   b       @L1
@STP:
   b     @exit
]


# define KM kMaxInstructions
// rMask provides the correct constant for the rlwinm assembly instruction
# define rMask (                          \
   ((KM &  32) >> 5)*25+                  \
   ((KM &  64) >> 6)*24+                  \
   ((KM & 128) >> 7)*23+                  \
   ((KM & 256) >> 8)*22+                  \
   ((KM & 512) >> 9)*21+                  \
   ((KM &1024) >>10)*20                     \
)

static asm void Execute0(CRT_memory memory,UInt32 memUsed) {
// This version of Execute relies on kMaxInstructions to
// equal memUsed (= 1 << address_bits)
register int* memSpan;
register int* ACC;
register int* BASE;
register int* CI;
register int* M1;
register int* M2;
register int* PI1;
register int* PI2;
register int* instruction;
register int* CIinstField;
   fralloc

   rlwinm memSpan,addMask,2,0,29
   add     endPC,MEM,memSpan
```

```
  b      @theCode
@GetCodeBase:
  mflr   BASE
  li   ACC,0
  li   CI,1

@restart:
  rlwinm PC,CI,2,rMask,29
  addi   CI,CI,1
  add    PC,MEM,PC
  lwz    PI1,0(PC)
  lwzu   PI2,4(PC)
  b      @decode

@loop:
  lwz    PI1,4(PC)
  addi   CI,CI,2
  lwzu   PI2,8(PC)
@decode:
  rlwinm instruction,PI1,28,20,22
  rlwimi instruction,PI2,25,23,25
  add    instruction,BASE,instruction
  mtctr    instruction
  rlwinm addShft1,PI1,2,rMask,29
  rlwinm addShft2,PI2,2,rMask,29
  bctr

@theCode:
  bl   @GetCodeBase

#undef doLDN_STO
#define doLDN_STO                          \
  lwz    PI1,4(PC);                        \
  lwzu   PI2,8(PC);                      \
  lwzx     M1,MEM,addShft1;     \
  addi   CI,CI,2;                          \
  rlwinm instruction,PI1,28,20,22;\
  rlwimi instruction,PI2,25,23,25;\
  add    instruction,BASE,instruction;\
  mtctr    instruction;                  \
  neg    ACC,M1;                           \
  rlwinm addShft1,PI1,2,rMask,29; \
  stwx     ACC,MEM,addShft2;       \
  rlwinm addShft2,PI2,2,rMask,29; \
  bctr
#undef doLDN_SUB
#define doLDN_SUB                          \
  lwz    PI1,4(PC);                        \
  lwzu   PI2,8(PC);                      \
  lwzx     M1,MEM,addShft1;     \
  addi   CI,CI,2;                          \
  rlwinm instruction,PI1,28,20,22;\
  rlwimi instruction,PI2,25,23,25;\
  lwzx   M2,MEM,addShft2;       \
  add    instruction,BASE,instruction;\
  mtctr    instruction;                  \
  neg    ACC,M1;                           \
  rlwinm addShft1,PI1,2,rMask,29; \
  subf   ACC,M2,ACC;                       \
  rlwinm addShft2,PI2,2,rMask,29; \
  bctr
#undef doSUB_STO
#define doSUB_STO                          \
  lwz    PI1,4(PC);                        \
  lwzu   PI2,8(PC);                      \
  lwzx     M2,MEM,addShft1;     \
  addi   CI,CI,2;                          \
  rlwinm instruction,PI1,28,20,22;\
  rlwimi instruction,PI2,25,23,25;\
  add    instruction,BASE,instruction;\
  mtctr    instruction;                  \
  subf     ACC,M2,ACC;                   \
  rlwinm addShft1,PI1,2,rMask,29; \
  stwx     ACC,MEM,addShft2;       \
  rlwinm addShft2,PI2,2,rMask,29; \
  bctr

@JMP_JMP:doJMP1;nop10
@JMP_JRP:doJMP1;nop10
@JMP_LDN:doJMP1;nop10
@JMP_STO:doJMP1;nop10
@JMP_SUB:doJMP1;nop10
@JMP_SU2:doJMP1;nop10
@JMP_CMP:doJMP1;nop10
@JMP_STP:doJMP1;nop10

@JRP_JMP:doJRP1;nop10
@JRP_JRP:doJRP1;nop10
@JRP_LDN:doJRP1;nop10
@JRP_STO:doJRP1;nop10
@JRP_SUB:doJRP1;nop10
@JRP_SU2:doJRP1;nop10
@JRP_CMP:doJRP1;nop10
@JRP_STP:doJRP1;nop10

@LDN_JMP:doLDN(1);doJMP2;nop9
@LDN_JRP:doLDN(1);doJRP2;nop8
@LDN_LDN:doLDN(2);b @loop;nop13
@LDN_STO:doLDN_STO;nop3
@LDN_SUB:doLDN_SUB;nop;nop
@LDN_SU2:doLDN_SUB;nop;nop
@LDN_CMP:doLDN(1);doCMP2;b @loop;nop9
@LDN_STP:doLDN(1);doSTP2;nop9

@STO_JMP:doSTO1(1);doJMP2;nop3
@STO_JRP:doSTO1(2);doJRP2;nop;nop
@STO_LDN:doSTO1(3);doLDN(2);b @loop;nop5
@STO_STO:doSTO1(4);doSTO2;b @loop;nop6
@STO_SUB:doSTO1(5);doSUB(2);b @loop;nop5
@STO_SU2:doSTO1(6);doSUB(2);b @loop;nop5
@STO_CMP:doSTO1(7);doCMP2;b @loop;nop3
@STO_STP:doSTO1(8);doSTP2;nop3

@SUB_JMP:doSUB(1);doJMP2;nop9
@SUB_JRP:doSUB(1);doJRP2;nop8
@SUB_LDN:doLDN(2);b @loop;nop13
@SUB_STO:doSUB_STO;nop3
@SUB_SUB:doSUB(1);doSUB(2);b @loop;nop11
@SUB_SU2:doSUB(1);doSUB(2);b @loop;nop11
@SUB_CMP:doSUB(1);doCMP2;b @loop;nop9
@SUB_STP:doSUB(1);doSTP2;nop9

@SU2_JMP:doSUB(1);doJMP2;nop9
@SU2_JRP:doSUB(1);doJRP2;nop8
@SU2_LDN:doLDN(2);b @loop;nop13
@SU2_STO:doSUB_STO;nop3
@SU2_SUB:doSUB(1);doSUB(2);b @loop;nop11
@SU2_SU2:doSUB(1);doSUB(2);b @loop;nop11
@SU2_CMP:doSUB(1);doCMP2;b @loop;nop9
@SU2_STP:doSUB(1);doSTP2;nop9

@CMP_JMP:doCMP1;doJMP2;nop9
@CMP_JRP:doCMP1;doJRP2;nop8
@CMP_LDN:doCMP1;doLDN(2);b @loop;nop11
@CMP_STO:doCMP1;doSTO2;b @loop;nop12
@CMP_SUB:doCMP1;doSUB(2);b @loop;nop11
@CMP_SU2:doCMP1;doSUB(2);b @loop;nop11
@CMP_CMP:b @loop;nop15
@CMP_STP:doCMP1;doSTP2;nop9

@STP_JMP:doSTP1;nop9
@STP_JRP:doSTP1;nop9
@STP_LDN:doSTP1;nop9
@STP_STO:doSTP1;nop9
@STP_SUB:doSTP1;nop9
@STP_SU2:doSTP1;nop9
@STP_CMP:doSTP1;nop9
@STP_STP:doSTP1;

@exit:
  frfree
  blr

// If an exception (CI instruction bit set) was detected,
// execution continues here, with a loop which checks CI
// on every instruction.
@exception:
//reconstruct CI:
  addi   CI,CI,-1
  b      @theCode2
@getCodeBase2:
  mflr   BASE
```

```
@L1:
// test CI bits 13-15 before incrementing CI
// this appears to be what M1SIM.EXE does
   rlwinm.   CIinstField,CI,0,16,18
   addi        CI,CI,1
   rlwinm TEMP,CI,2,rMask,29
   lwzx        M1,MEM,TEMP    //load instr word from memory
   beq-        @L2
   cmpwi     CIinstField,0x2000
   bne-    @exit                    // return if CI instruction != 1
   add     PI1,PI1,M1          // add instruction to instruction !!
   b           @L3
@L2:
   mr          PI1,M1              // the normal case : PI = M1
@L3:
   rlwinm      instruction,PI1,23,25,27     // instruction * 16
   add     instruction,BASE,instruction
   mtctr       instruction
   rlwinm addShft1,PI1,2,rMask,29
   bctr

@theCode2:
   bl   @getCodeBase2
@JMP:
   lwzx CI,MEM,addShft1
   b       @L1;       nop;nop
@JRP:
   lwzx    TEMP,MEM,addShft1
   add     CI,TEMP,CI
   b       @L1;       nop
@LDN:
   lwzx    TEMP,MEM,addShft1
   neg     ACC,TEMP
   b       @L1;          nop
@STO:
   stwx    ACC,MEM,addShft1
   b       @L1;       nop;nop
@SUB:
   lwzx    TEMP,MEM,addShft1
   subf    ACC,TEMP,ACC
   b       @L1;          nop
@SUB2:
   lwzx    TEMP,MEM,addShft1
   subf    ACC,TEMP,ACC
   b       @L1;          nop
@CMP:
   cmpwi   ACC,0
   bge     @L1
   addi    CI,CI,1
   b       @L1
@STP:
   b    @exit
}

pascal void ExecuteBabyProgram(
   CRT_memory memory,
   UInt32 address_bits) {
   UInt32 memUsed=1<<address_bits;
   if (memUsed>kMaxInstructions)
      return;
// copy of memory allocated on procedure stack
   UInt32 memCopy[kMaxInstructions+2];
   CopyMem(memCopy,memory,memUsed);

// 2 * STP as sentinel:
   memCopy[memUsed]=-1;
   memCopy[memUsed+1]=-1;

   if (memUsed==kMaxInstructions)
   Execute0(memCopy,memUsed);
   else Execute(memCopy,memUsed);
// return processed memory to caller
   CopyMem(memory,memCopy,memUsed);
}
```

MT

by Mike Morton

Delayed Messaging

Benefits of Procrastination: Delayed Messaging using the Foundation Kit

Introduction

When you send an Objective-C message to an object, you expect that object to receive the message and process it immediately. Right? Maybe not...

The Foundation Kit provides a way to post messages and have them delivered later, through the method **performSelector:withObject:afterDelay:**. This method is part of the NSObject class, from which most other classes descend. In this article, we'll discuss not only how you use this method, but give some examples of how delayed messaging solves some difficult problems.

If you've just begun working with Foundation Kit and AppKit, don't worry – we'll explain things step-by-step. In each of the four examples below, we'll describe the problem, show the solution working in the application, and then review the relevant parts of the code. But first, let's look at the method itself.

How to Send a Delayed Message

The method which lets you send delayed messages to any NSObject is declared like the following

```
- (void) performSelector
  :(SEL) aSelector
  withObject :(id) anArgument
  afterDelay :(NSTimeInterval) delay;
```

This method takes three arguments:

aSelector specifies the message to send. A selector is sort of the "name" of a method, a very distant cousin of C's function pointers. (In some implementations, a selector is just a "char *" pointer to a unique string value, and this can be useful in debugging, but you should never depend on it being so.) Unlike function pointers, selectors have their own data type, SEL, and you can refer to them in Objective-C code using the @selector(...) construct. For example, if you'd like to send the message **setFoo:**, you can specify the selector for that message with @selector(setFoo:).

anArgument is an optional object to send with the message. If you don't want to pass anything, you can pass *nil*. But keep in mind that if you do want to pass something, it must be an object – not an integer or other C type. (In theory, the method specified by *aSelector* should take a single argument, but no-argument methods seem to work fine. Of course, two-argument methods don't work, because there's no way to pass a second argument with this API.)

delay is the number of seconds to delay before sending the message. [If you used the method like this one in earlier releases of Foundation, note that the delay is no longer expressed in milliseconds.] If the program is busy, the message may take longer before being delivered, but it will never get delivered early.

Using this method, you can send any one-argument message with a delay. For example, if you have an object printer which implements the method **printString:**, you can have it receive a **printString:** message after a delay of ten seconds with this code:

```
[printer performSelector :@selector(printString:)
  withObject :@"Hello, world!"
  afterDelay :10.0];
```

Mike Morton <mike@mikemorton.com> spent seven years developing software for Macintosh, then seven years developing for NextStep and OpenStep. (You may draw your own parallels to biblical dreams of seven lean years and seven fat years.) He currently works for Apple Enterprise Technical Support, which old-timers know as "NeXT Tech Support".

This is almost the same as messaging the object directly with [printer printString :@"Hello, world!"], except that the message gets delivered later, not "while you wait". Also, of course, if **printString**: returns a value, you can't get that value when using delayed messaging, because you want to continue before the message even gets received. (Modern CPUs are fast, but still don't support time-travel.)

If you change your mind about a delayed message you sent earlier, you can cancel it, using an NSObject class method:

```
+ (void) cancelPreviousPerformRequestsWithTarget
  :(id) aTarget
  selector :(SEL) aSelector
  object :(id) anArgument;
```

That's the "how" of delayed messaging. But why would you want to use it. Each of the four examples shows one reason why – let's take a look in more detail. The source code for each of these demos is available online at ftp.mactech.com.

Example #1: Are They Going to Click Again?

Suppose you're implementing a web browser (just to give Netscape and Microsoft some competition). If the user clicks on a link, the browser displays the new page in the same window. But if they double-click the link, you want to open a new window.

What should your program do on that first click? You can't display the new page (because if a second click shows up, that means the user didn't want to change this window's display). But you also can't ignore the first click (because if no second click arrives, you want to display the new page).

Try it: The demo doesn't browse the web, but it does show an example of not acting on every click. Quickly click on the red rectangle one or more times; it changes color with every click. Now check "Wait for clicks to stop before redisplaying", and it will change color only after you finish clicking.

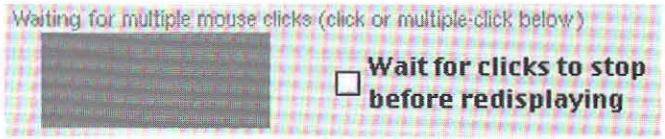


Figure 1. *Acting on multiple clicks.*

This part of the demo uses a custom view named ColorView, a subclass of the AppKit's important NSView class. The two instance variables for the class remember (1) what color it currently draws, and (2) how quickly it redraws. The class interface, from ColorView.h, begins like this:

Listing 1: Start of ColorView class interface

```
@interface ColorView : NSView
{
  NSColor *color;          // color we fill with
  BOOL  delaysDisplay;     // YES => wait to redraw
}
```

The header file also lists *access methods*, methods which get and set the instance variables, but these aren't shown in Listing 1.

The implementation, in ColorView.m, overrides some methods from NSView – **initWithFrame:**, which all views use to initialize; **drawRect:**, which displays the view's contents; and **mouseDown:**, which handles mouse clicks. The latter two are more important.

The **drawRect:** method (Listing 2) is simple. It sends a **set** message to the view's current color, so that subsequent PostScript drawing will use that color, and calls a PostScript function to fill in the rectangle.

Listing 2: ColorView's drawRect: method

```
- (void)drawRect          // redraw contents of...
  :(NSRect) aRect;        // INPUT: ...this rect
{
  [[self color] set];     // paint with this color
  PSrectfill (aRect.origin.x, aRect.origin.y,
        aRect.size.width, aRect.size.height);
}
```

The **mouseDown:** method (Listing 3) is also short, but does a lot more. It updates the view's color, based on the event's *click count* – the NSEvent method **clickCount** returns 1 for a click, 2 for a double-click, etc.

Listing 3: ColorView's mouseDown: method

```
- (void) mouseDown
  :(NSEvent *) theEvent;
{
 // Update our color: red for a single-click,
 // orange for a double-click, etc.
  [self _setColorAtIndex :[theEvent clickCount]-1];
}
```

When **mouseDown:** sends **_setColorAtIndex:**, that method winds up calling **setColor:**, which is where things start to get interesting. When you give the ColorView a new color, it will redraw itself, but it may not do so immediately. After it remembers the new color, it checks "if ([self delaysDisplay])". If it doesn't delay displaying, it sends "[self setNeedsDisplay :YES]" – that's how you ask any NSView to redraw.

But if you click the checkbox "Wait for clicks to stop before redisplaying", the checkbox sends **takeDelaysDisplayFrom:** to the ColorView, which will take a new value of delaysDisplay from the checkbox.

And when [self delaysDisplay] returns YES, the ColorView knows that it shouldn't redraw, but it also that it shouldn't forget about the color change, since it needs to redraw at some point. So it does the code in Listing 4: first, it removes any pending redraw requests, then it sends a **_setNeedsDisplayYes** message, but delaying the delivery. (Why doesn't it send a delayed **setNeedsDisplay:**, with an argument of YES? Because the argument of a delayed message must be an object.)

Listing 4: ColorView's delaying of redraw

```
{
  // Cancel any pending message...
    [NSObject cancelPreviousPerformRequestsWithTarget
              :self
       selector :@selector(_setNeedsDisplayYes)
        object :nil];

  // ...and queue a new one
    [self performSelector
             :@selector(_setNeedsDisplayYes)
      withObject :nil
      afterDelay :REDRAW_DELAY];
}
```

How on earth does this work? Every time you click the ColorView, it remembers a new color, and posts a delayed message to redisplay itself soon. If you click again, soon enough, it cancels that delayed message and posts another, and so on. When you stop clicking, the last message you posted will get delivered, and the view redraws just once.

It's sort of like being a painter in a hotel with a finicky owner. Every few minutes, the owner changes their mind about what color they want. So you call the front desk, and say "Wake me in an hour". If the owner wakes you up early to change the color again, you phone the front desk and say "Cancel that wake-up call, and wake me an hour from now". Only after the owner goes a full hour without changing their mind will you get a wake-up call (a delayed message) – and then it's time to paint.

Without delayed messaging, it would be very hard to "predict" whether a mouse click will be followed by another. With the ability to post a message for future delivery, it takes just a few lines of code.

Example #2: Are They Done Typing?

The world is full of query user interfaces in which you type what you want to find, then press Return (or click "OK" or "Find" or "Search" or...). But if doing a search doesn't take a lot of time, why make the user press return? Why not just search when the users pauses in typing?

Try it: Type one or two letters in the box labeled "Find:". A moment after you finish typing, the "Items found" scrolling list will change to list only entries containing what you typed. Type some more to refine the search, or delete characters to expand it again. Adjust the "Wait for pause of..." field to find a delay which works well for you.

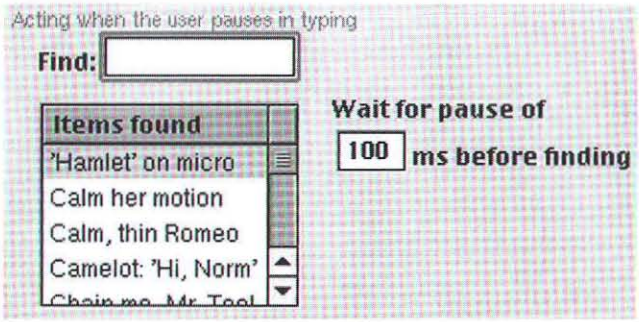


Figure 2. *Automatic querying.*

This solution works much like the previous one, but instead of waiting for a pause after one or more mouse clicks, you're waiting for a pause after one or more keystrokes (or other changes).

We can't just override **keyDown:** like we did for **mouseDown:** in the last example, because the field can change for things other than keystrokes (such as by pasting or cutting text). But it's easy to monitor an NSTextField for changes: we make the application controller be the *delegate* of the field. (You can see this relationship by inspecting the field's connections in the nib file.)

Each time the user changes the field, the controller receives a **controlTextDidChange:** message. [NSTextField documentation in the DR2 release promises a **textDidChange:**, but this seems not to get sent, and NSControl documentation says it's **controlTextDidChange:**.] The application controller's implementation is in Listing 5. It wipes out the results of any previous search, cancels any previous, delayed **reload** messages, and sends a delayed **reload** message to the table.

Listing 5: Handling a change in a query field

```
- (void) controlTextDidChange
  :(NSNotification *) notification; // IGNORED
{
 // Wipe out any cached list of filtered objects.
   [self setFilteredStrings : nil];

 // Cancel any pending message.
   [NSObject cancelPreviousPerformRequestsWithTarget
           :findResults
     selector :@selector(reloadData)
      object :nil];

 // Now tell the table to reload after the a delay.
 // (When it reloads, it'll get a new list of strings.)
   [findResults performSelector :@selector(reloadData)
          withObject :nil
          afterDelay :[findThreshhold floatValue]
                /1000.0];
}
```

Let's skip the details of how **reload** redisplays the table to show only items matching your search string, but — in brief — it goes like this:

- the table gets the delayed **reloadData**
- the table sends **numberOfRowsInTableView:** to the controller
- the controller computes [self filteredStrings]
- **filteredStrings** finds and saves matching strings
- the table sends **tableView:objectValueForTableColumn:** to the controller
- the controller uses the filtered strings to find the object value

One other thing to note: the method **awakeFromNib** also sends **reloadData** to the table, which makes the table display its contents when the application starts, before you type anything.

Example #3: Did They Let Go of that Slider Yet?

All controls give you a choice of when you'd like them to send you their *action message*. For a button, you can ask to get the message continuously, as long they hold the mouse down in the button (useful for a scroller's arrow) or only when they release the button (useful for an "OK" button). You can set which way a control behaves with the **setContinuous:** method.

Suppose yourÄ3-D drawing application has a slider used to set the viewing angle. While they're dragging the slider, you want to quickly draw a wire-frame image to give them an idea of what it'll look like. When they release the mouse, you want to redraw a full image.

But the slider sends only one type of action message, during dragging or when you release. How can you act on both"

If you want to do it the hard way, you could subclass NSSlider to support two types of action messages or to tell you if it's currently tracking. If you want to do it the elegant way, you could ask the shared NSApplication instance for **currentEvent**, which is the last event processed, and ask what kind of event provoked the action message. But this is an article about delayed messaging, so we'll do it the fun way.

Try it: The demo doesn't do wire-frame drawing, but as you drag the slider it does show you the "Sliding" value, then when you release the mouse, it'll show you the "Final" value.

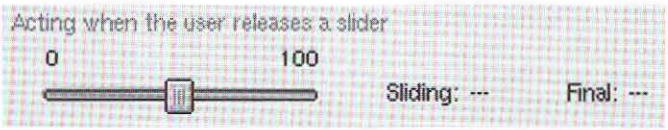


Figure 3. *Distinguishing dragging from mouse-up.*

This solution is a little tricky, and illustrates something about when delayed messages are actually delivered. As we've said before, a delayed message will never get delivered early, but it may get delivered late. This is because the application processes delayed messages only when it's idle. During the time you're dragging the mouse, the application is busy tracking the drag, and doesn't check if it needs to deliver delayed messages. So you can post a delayed message which says "the mouse went up", trusting that it won't get released until the user releases the mouse, and the control finishes tracking. (Nothing in the documentation promises this will work, so don't use this idea in a production application. But for demo purposes, it's fine.)

Listing 6 shows the implementation is simple: The slider sends **mouseUpSliderAction:** when you move it, and this method will:

(1) update the "Sliding" value in the UI
(2) wipe out any left-over "Final" value in the UI
(3) post a delayed **_sliderFinished** if it hasn't done so already.

Listing 6: The action method for the slider

```
- (IBAction) mouseUpSliderAction :sender;
{
 // Display "sliding" value, and clear "final" value
  [mouseUpSlidingValue setIntValue
    :[mouseUpSlider intValue]];
  [mouseUpFinalValue setStringValue :@"--"];

 // If this is our first time getting this message
 // since they began dragging, we want to remember
 // to do something when they release the mouse, too.
 // If we haven't queued that reminder yet, do it now.
  if (! mouseUpMessagePending)
  {
    [self performSelector :@selector(_sliderFinished)
          withObject :nil
          afterDelay :0];
    mouseUpMessagePending = YES;
  }
}
```

As you might guess, the **_sliderFinished** method clears the "Sliding" value, sets the "Final" value, and clears the flag. That's all there is to it.

Example #4: Simple Animation

You're almost done with that prototype which you want to sell to Microsoft, and it's got everything a Microsoft product should have: a complex UI, incompatible functionality, too many features, bugs galore... what else could Microsoft want?

Then you remember: They want a little assistant in the corner of the screen, animated to scratch or lick itself when you're not doing much. How can you animate when you don't have a real-time system — and how can you make sure the animation doesn't slow things down when the user is working?

If you've read this far, I hope you've guess the answer by now: you can do animation with delayed messaging.

Try it: Click "Run" to start the selection running around the matrix. Try typing different animation rates into the "Animate ... frames/second" field — you can type a new value while the animation is running.

(Now try dragging the slider (from example 3). The animation doesn't run while you're dragging. Again, delayed messages don't get delivered at all during certain operations.)

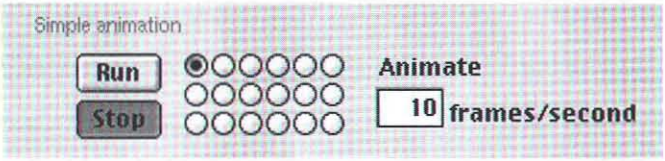


Figure 4. *Simple animation*

The code to implement this depends on one state variable, animationDirection, which keeps track of which way the selection is moving in the matrix. Clicking "Run" sends the **animationRun:** message, which just makes sure the direction is initialized, then sends **_doAnimation** to do the first "frame".

The **_doAnimation** method (shown in Listing 7) draws one "frame" by finding out what cell is selected and selecting the next one, reversing direction if it's reached the last cell. Then it calculates the current inter-frame delay from the text field, and queues a message to do itself again after that delay.

Listing 7: Drawing one "frame" of animation

```
- (void) _doAnimation;
{
  int    tag;
  float  delayInSeconds;

 // Get current position; bump it in current direction
  tag = [animationMatrix selectedTag];
  tag += animationDirection;

 // Cheap hack: If the tag runs off the end,
 // there's no cell with that tag.
  if ([animationMatrix cellWithTag :tag] == nil)
  {
  // Whoops! Switch direction and head the other way.
    animationDirection = -animationDirection;
    tag += (2*animationDirection);
  }

 // "Animate" to the new position. Because it's a set
 // of radio buttons, we needn't deselect the old cell.
  [animationMatrix selectCellWithTag :tag];

 // Figure the delay, which may have changed if they
 // typed a new number, and queue a message to
 // do all this again.
  delayInSeconds = [animationDelay floatValue]/1000.0;
  [self performSelector :_cmd
        withObject :nil
        afterDelay :delayInSeconds];
}
```

Instead of Äusing @selector(_doAnimation) as the selector, I chose to pass just **_cmd**. This obscure variable is automatically declared in every Objective-C method; it refers to the method currently being executed. Using it instead of the @selector(...) construct is largely a stylistic choice — I wanted to make it clear that the method is invoking itself.

But... wait a minute! If a method invokes itself unconditionally, doesn't that cause infinite recursion? Usually, yes. But in this case, the invocation doing the calling returns before it gets called again. So each invocation of the method happens independently of the previous ones — and without the previous ones on the stack.

So it's not recursive, but just like in a correct recursive algorithm, there is an end to the succession of invocations. If you click the "Stop" button, the **animationStop:** method will use **cancelPreviousPerformRequestWithTarget:selector:object:** to cancel the currently-pending invocation, breaking the chain and stopping the animation.

The Foundation Kit's NSTimer class provides another convenient way to do animation by repeatedly sending a message for you at fixed intervals. I chose delayed messaging here because it makes it easy to change the animation rate: note how **_doAnimation** immediately reacts to a new rate in the field, while it would be a little more complicated with a timer.

Again, keep in mind that delayed messages get performed only when the application is idle. You can have multiple animations running, but they'll take up no time during intensive computations. This can be useful if your UI needs to flash to show the user important conditions, but doesn't want to slow things down.

Other Applications

- If updating an inspector or ruler slows your application, send it a delayed message. If the user changes the selection again soon, the inspector or ruler can avoid an unnecessary update.

- You can minimize propagation in a two-level network. If multiple objects can trigger some reaction (such as recalculating) in a single target object, they can all send delayed messages with a zero delay, cancelling previous messages, and the target will receive just one message.

- You can make an app terminate after a given period of inactivity by sending a delayed **terminate:** message to the application, then cancelling and re-posting the message each time there's some activity.

- ...watch this space! I'd like to hear of uses you find for delayed messaging.

Conclusion

Delayed messaging is a simple Foundation Kit feature which can help solve some challenging problems. Keep the technique in mind, and you'll find interesting applications for these and other techniques.

Acknowledgements

Thanks to Art Isbell and Byron Han for their comments and suggestions. Thanks also to Lee Worden, for his help and ideas on an earlier version of this article, written for the first release of the Foundation Kit.

Further Reading

In addition to the NSObject class description, and its brief description of delayed messaging, you might find the following interesting:

NSObject class documentation

In addition to providing the authoritative word on the methods describes above, this also explains **performSelector:withObject:afterDelay:inModes:**, which gives you finer control over how idle the application has to be before the message gets sent.

NSTimer and NSInvocation class documentation

Delayed messaging uses concepts from timers and invocations. While you need not understand either of these classes to use it, you might find the details enlightening.

Apple's documentation for Enterprise Objects Framework (EOF)

EOF uses delayed messaging internally in a variety of ways to improve efficiency. For example, certain notifications are delayed so redundant notifications can be coalesced into one.

Ben Shneiderman, *Dynamic Queries for Visual Information Seeking*, originally published in IEEE Software 11, 6 (November 1994), 70-77, available at <ftp://ftp.cs.umd.edu/pub/hcil/Reports-Abstracts-Bibliography/93-01html/3022.html>.

This article argues out that hardware has become so fast that you can evaluate queries instantly (without waiting for the user to press Return or click "OK" or...), and – unlike so many articles on GUIs — has a number of figures illustrating specific GUI ideas.

MT

by Frank Vernon and Dave Mark,

A PowerPlant Update, Part 3

This month's column is the third installment in our series on PowerPlant. In the past two months, we heard from two key members of the PowerPlant team, Greg Dow and John Daub. This month, we'll hear from another key member of the team, Frank Vernon.

Frank Vernon has been working for Metrowerks since CW9. Before joining Metrowerks, Frank worked at Apple doing R&D for first eWorld and later Internet Services in their Boulder, Colorado group. When Frank joined Metrowerks, he took over the networking classes from Eric Scouten and completely rewrote them for CW11. The first of the Internet Classes were introduced around CW9 as well. Stuffit classes and Internet Config classes were added right after CW11.

Frank also worked on the 1.1 update to Marionet for Allegiant Software (SuperCard, etc.). Marionet is a faceless background application that utilized Apple events. It supported SMTP, POP3, NNTP, FTP, and a custom chat client/server.

In addition to his engineering background, Frank also has a degree in music.

Dave: Tell me a little bit about how you got started with PowerPlant.

Frank: Before MW I was doing R&D for Apple in a small autonomous group based in Boulder, Colorado. Our group initially began working on eWorld and later evolved into the Internet Services division as eWorld was being dismantled. This was an interesting R&D group to be involved with in that we did simultaneous technology R&D and business modeling. We were focused on creating technology directly applicable to the business models we thought would be successful rather than being a "pure" R&D technology group. I personally found this type of business and results driven R&D to be quite rewarding.

After leaving Apple in early 1996 and while at MacWorld San Francisco I heard a rumor that Eric Scouten, the original author of the PowerPlant Networking Classes, was looking for someone to take them over so he could concentrate on Constructor full-time. I eventually got in touch with Eric and began working on the classes. An interesting story here is that I still, to this day, have never met Eric face to face. Despite having worked relatively closely with him for almost 2 years, as I first started taking over the classes and then began rewriting them we have never met in person. We played "booth tag" that year while at MacWorld and have continued to miss each other at subsequent MacWorlds and Developer Conferences. The majority of our work was done via Email and occasionally we would have a phone call to hash out the nastier details.

Dave: Tell us about the evolution of the networking classes.

Frank: Long before working at Metrowerks, Eric Scouten had created a very nice set of MacTCP based networking libraries called TurboTCP. When OT was first being released, he had the idea to abstract MacTCP and OT into a single library so developers would not have to deal with developing seperately for both in their applications. Since he was working at Metrowerks at the time, he created them as PowerPlant classes. Being heavily involved in Constructor, however, he didn't have time to really get them into shape. In fact, when I took them over for the CW9 release they were still in the PowerPlant "In Progress" folder.

When I took over the Networking Classes the basic groundwork had been laid, but they did have several major limitations. The first thing I added was "Listen" support. This allowed them to accept incoming

connections so they could be used for network servers. Prior to this they operated in client mode only. I was also fixing lots of bugs and preparing them for release outside of the "In Progress" folder.

Abstracting OT & MacTCP into a common interface was an interesting prospect to say the least. Besides the obvious complexity of trying to make the two disparate stacks behave in the same way, we also encountered numerous little problems. Not the least of which was simply testing them.

For each feature or bug we encountered they had to be tested in all possible combinations of MacTCP and OT, 68K and PPC, Asynchronous mode and Threaded mode, and with and without Interrupt notifications. This alone made for many long and frustrating nights of debugging. As anyone who's done network programming can tell you, even on a good day, debugging network code can be like searching for a needle in a needle factory.

In addition, Eric's initial designs were quite ambitious. Not only were we trying to support two radically different TCP/IP stacks through a single interface, but we were also attempting to address all of the common network programming styles such as asynchronous and threaded operations while also maintaining an interrupt driven notification model. These interrupt driven notifications were particularly difficult to manage due to their memory allocation limitations and debugging restrictions. While not impossible to comprehend, most developers simply didn't want to deal with all the complexity that these disparate paradigms introduced into the classes. After going through several releases we had a lot of feedback that they were simply getting too complex for the needs of most developers.

After CW10 Eric and I worked out a major overhaul to the classes which is what you see today in the Networking Classes. From the feedback we received we decided to streamline the classes and make them as stable and as easy to use as possible while still maintaining the initial design requirement of abstracting MacTCP and OT into a single interface.

One of the major advantages that we had in redesigning the classes was that the Thread Manager was also reaching maturity. Prior to this time, most network programmers used heavily asynchronous, event driven, programming models. Without having multiple threads available you simply couldn't block and wait for a network event to complete. This could tie up your application or even the entire machine for each call which of course meant that your interface and the whole machine would be unresponsive while you waited for completion.

By settling on a threaded model for the new classes we were able to make several major improvements. The present classes, for instance, hide 99% of the issues related to interrupt driven events and associated memory and debugging issues from the developer. Also, in my mind, the threaded model of network programming is much easier for casual network programmers to understand and work with.

The threaded model allows for radically simplified state machines with most higher level protocols. Code that might have been thousands of lines before could often be reduced to hundreds.

Subtle side effects of this model also allow for more sophisticated objects like the current LInternetAddress object. This object now transparently translates between canonical DNS and IP dotted decimal style addresses. Without the straight-forward flow control of the threaded model this would not possible.

For example, the simple act of converting an IP address into a canonical host name in the past would have required a multi-stage state machine to negotiate the DNS interaction to resolve the address. In the present Networking classes you can simply do the following:

```
LInternetAddress theAddress("208.226.102.11");
Str255 theConnonicalName;
theAddress.GetDNSDescriptor(theConnonicalName);
```

Dave: What about support for other networking protocols such as AppleTalk within the Networking classes?

Frank: For strictly historical reasons, the initial concept behind the Networking classes was only to abstract MacTCP and TCP/IP aspects of OT into a common library. We do receive sporadic requests to support other things like AppleTalk and some of the more sophisticated OT features from time to time. With Apple in the midst of major updates around Carbon and beyond we are in the process of considering our options for the future. We are actively seeking input from the developer community in this matter. As always, we encourage people to contact Metrowerks tech support with any suggestions and requests they have along these lines.

Dave: Speaking of the future, what about Apple's planned changes to a Sockets-based networking model? How will this effect the Networking classes?

Frank: As I mentioned, we are still in the process of determining how to move forward towards Carbon and beyond. I can tell you however that Metrowerks recognizes the importance and growing ubiquity of Internet protocols and networking in general to modern applications. We are going to do our best to support whatever paradigms Apple chooses moving forward.

Dave: Can you describe the difference between the Networking classes and the Internet classes?

Frank: Based of course on the Networking Classes, the Internet Classes address specific Internet protocols. Presently these classes support: HTTP, SMTP, POP3, FTP, NTP, FINGER, and IRC. In addition, they support associated standards such as Internet Config, MIME, MD5, BinHex, UUEncode/UUDecode, MacBinary II, URL's, and others.

The design philosophy behind the Internet protocols was to represent the protocols as faithfully as possible while also supplying simple interfaces. For instance, many protocols have interfaces such as LFTPConnection::PutFile which take all the necessary parameters such as the name of the remote host, username, password, the name of the file in question, etc. and simply send or retrieve the files. The developer doesn't need to know much if anything about the protocol itself. If however they need to do more sophisticated procedures they also have complete control over the protocol at each step in the process.

As much as possible I've also tried to leverage other features of PowerPlant to keep the usage of the Internet Classes familiar to anyone who has used PowerPlant. For example, I use the standard Broadcast/Listen mechanism for progress and status reporting during the protocol execution. This allows for easy and familiar interface creation just like any other control.

The Internet classes also have a few nice features like the transparent use of temporary file buffering to allow the sending and receipt of arbitrarily large files (up to available disk space.) This buffering mechanism (represented in LDynamicBuffer) will automatically and transparently make decisions about either keeping data in memory or swapping out to temporary files when necessary. This is all hidden behind standard PP Stream classes but the developer never need know if the data is in a file on disk or resident in application memory. This greatly simplifies things such as large FTP file transfers.

In addition to the simplified interfaces available with the protocols themselves, I have also recently added a new utility class called UInternetProtocol. These utility methods even further simplify the process by simply taking a standard URL and doing all the work behind the scenes. These methods will automatically do things such as creating a new thread, parsing the URL, and resolving the URL. These are especially useful for adding simple URL based Apple event support to any application. (Little

known tip: The Internet Example application also demonstrates use of the GURL and FURL Apple events and includes the AETE resources. Try setting the Internet Example Application as your default FTP client in Internet Config and click on an FTP URL in your browser.)

UInternetProtocol FTP Example:

```
LURL theURL("ftp://user:pass@host/filepath");
UInternetProtocol::FTP_GetFile(nil, theURL, nil);
```

These two lines are all that are necessary to get the FTP file at the host specified using the name and password imbedded in the URL. If no username and password are included then anonymous login will be attempted.

Wherever possible defaults from Internet Config will be used for all the UInternetProtocol methods. This includes download file directories, default servers, usernames, passwords, email addresses, etc.

Dave: What can you tell me about the StuffIt classes?

Frank: The StuffIt classes grew out of a number of requests we had to somehow integrate file compression and decompression into the Internet classes. The people at Aladdin were quite helpful in allowing us to include the API to the StuffIt engine with the CodeWarrior releases.

What I attempted to do with these classes was to simplify the StuffIt Engine API by making assumptions about some of the more common usage. For the most part these also hide a lot of the necessary API requirements about properly opening and closing the engine, handling error conditions, and properly navigating archives. I've also attempted to represent the Engine's API into OOP methodologies so C++ developers could easily integrate its usage into their applications.

UStuffItSupport Example:

```
void
StuffItExample::StuffItGetFile()
{
//Do a StandardGetFile for the file to Stuff
StandardFileReply theReply;
::StandardGetFile(nil, -1, nil, &theReply);
if (!theReply.sfGood)
   return;

//Stuff the file assuming the same file name, etc.
try {
   UStuffItSupport::StuffFile(theReply.sfFile);
} catch (OSErr err) {
   DisplayStuffItError(err);
}
}
```

By comparison, the direct interface to the StuffIt Engine API to stuff a file is:

```
extern pascal OSErr  StuffFSList (long magicCookie,
                        FSSpecArrayHandle fileList,
                        FSSpecPtr archiveSpec,
                        FSSpecPtr resultFSSpec,
                        Boolean compressOriginal,
                        Boolean deleteOriginal,
                        Boolean encryptOriginal,
                        Boolean resolveAliases,
                        Boolean noRecompression,
                        short conflictAction);
```

As you can tell, the StuffIt Classes go a long way towards simplifying the common usage of the StuffIt Engine.

Another class, LstuffItArchive, allows you to work directly with StuffIt Archive files. With this class you can walk an archive, insert or delete files from an archive, create new archives, etc..

By the way, I should mention that a license from Aladdin is still required to ship the StuffIt engine with an application.

Dave: I know you've done some work on email servers. What kinds of changes have you seen recently in email server technology?

Frank: Actually, there have not been a lot of significant changes in email server technology for quite a while. The basic protocol, SMTP, has remained virtually unchanged for quite some time. While extensions are added to SMTP from time to time, it has generally proved to be a very effective, if not sophisticated, protocol. I believe in fact that its lack of sophistication is probably directly responsible for its longevity. (A lesson from which we could probably all learn something.)

The primary change is that you are now seeing more and more customized servers acting as gateways into other non-SMTP based systems such as Lotus Notes.

In addition, many of the older servers were never designed to deal with the larger volumes of mail you now encounter on the rapidly growing Internet. Many of the larger ISPs for instance have a lot of problems simply with volume. These older mail servers often require either complex reconfigurations or complete rewrites to handle high volumes.

We're also beginning to see more task specific email servers. Servers that focus on mailing list management and other automated mail handling tasks.

There are lots of startup companies out there dealing with email issues. Things like automated mail responders for customer support, customized email reminder services, and purchase transaction systems just to mention a few.

Email is arguably the most popular protocol on the Internet. It's the original "Push" technology. It's strange in way that only now are people really concentrating on email as "the" way to interact directly with customers.

MT

by Jeff Clites <online@mactech.com>

Advanced C++ Algorithms

Algorithms are at the very heart of programming. Fortunately, the C++ standards committee saw fit to include an algorithms library to work in conjunction with the container classes of the STL. Although immensely useful, this library really covers only the most commonly used, general purpose tasks, and for specialized applications you have to look elsewhere. This month we are going to take a tour of some powerful algorithm libraries that pick up where the STL leaves off.

Algorithms in General

There are a number of printed references for learning about algorithms and the theory behind them. *Algorithms in C++*, 3rd ed., by Robert Sedgewick, is a new version of a classic reference, full of lucid explanations and copious illustrations, and there is a freeware program, MacBalsa, which animates many of the algorithms from the book. *The Art of Computer Programming* series by Donald E. Knuth is the definitive work on almost all things computing, and it is referenced by just about any writing which mentions the word "algorithm." *Numerical Recipes in C* by William H. Press, et al, is another classic, with an emphasis on practicality, and the full text is available in pdf form for preview on the web. On the other hand, a page at JPL warns against using the code from Numerical Recipes as-is, so you might want to check out these sites in tandem. For your general algorithm needs, check out Netlib, Codepage, and Programmer's Oasis—they have descriptions and links to a plethora of code sources.

MacBALSA
<http://www.eg.bucknell.edu/~zaccone/MacBALSA/MacBALSA.html>
Don Knuth's Home Page
<http://www-cs-staff.stanford.edu/~knuth/>
Numerical Recipes Home Page
<http://www.nr.com/>
Why not use Numerical Recipes?
<http://math.jpl.nasa.gov/nr/>
Netlib
<http://www.netlib.org/c++/>
{Codepage 2.2}
<http://www.iro.umontreal.ca/~ratib/code/>
Programmer's Oasis: Algorithms, Data Structures
<http://www.utu.fi/~sisasa/oasis/oasis-algo.html>

Finite-State Machines

Finite-State Machines (or FSMs for short) are based on a mechanical model of computing, originating with the Turing Machine. Although conceptually simple, they are the natural way to express many computational processes, such as string searching or parsing. Object Mentor has code available for an FSM compiler, which allows you to define an FSM with a simple notation and then compile it into code for a set of C++ classes. (Also be sure to check out the cool "Game of Life" applet, Wator, on their home page.) Conveniently, Michael Schürig has done the work of preparing a Mac version of the compiler as a droplet, and it is posted on his web site.

Object Mentor Freeware by Email
<http://www.oma.com/Offerings/catalog.html>
Michael Schürig's Home Page
<http://www.schuerig.de/michael/>

Genetic Algorithms

Genetic algorithms are a class of algorithms for solving minimization or best-fit problems, and are especially useful for situations in which classic optimization techniques fail. They are so-named because their basic strategy mimics biology: to find the optimal solution, generate a pool of candidate solutions, take a selection of the best ones and "combine" these to generate a new pool of candidates, and repeat. Unlike other optimization techniques, their successful use often depends heavily on finding the right way to encode the problem, but if used correctly they can succeed despite the presence of local minima which plague other approaches. MIT's GAlib makes it easy to implement a variety of different genetic algorithm strategies, and TimGA is a program which graphically demonstrates the optimization process and lets you experiment with how various parameters effect performance. TimGA is PowerPlant-based, and the code is available online.

GAlib: Matthew's C++ Genetic Algorithms Library
<http://lancet.mit.edu/ga/>
TimGA Source Code
<http://hyperarchive.lcs.mit.edu/cgi-bin/NewSearch?key=TimGA>

THE Future

HERE AND NOW!

MACWORLD Expo/San Francisco

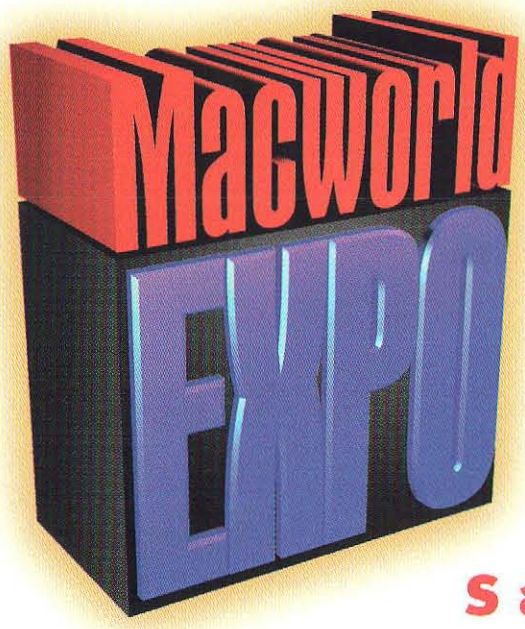
EXHIBITS: January 5-8, 1999

WORKSHOPS: January 4, 1999

MACWORLD/PRO CONFERENCE:
January 5-7, 1999

MACWORLD USERS CONFERENCE:
January 6-8, 1999

Moscone Convention Center



San Francisco, CA

For millions of computer users, Macintosh is synonymous with increased productivity and innovative technology advances. Come to San Francisco—home to a "think different" community—for the latest technologies and trends that will change the way you work, play and learn now and in the future.

At MACWORLD Expo/San Francisco, you'll demo, touch, see and feel all the hot new products—from more than 400 leading vendors. Learn time-saving productivity secrets through in-depth workshops, advanced and user-level conference programs. Hear from the brightest stars who are shaping the future of the Mac universe at the much-anticipated keynote address.

MACWORLD Expo/San Francisco has the latest solutions for:

- *the Internet*
- *digital content creation, management and delivery*
- *multimedia*
- *software development*
- *small business*
- *remote worker programs*
- *connectivity*
- *graphic design*
- *publishing*
- *education, research and development*

Take advantage of money-saving specials when you buy products right on the show floor.

Discover what the future holds at MACWORLD Expo/San Francisco. Register to attend today!

VISIT: www.macworldexpo.com

CALL: 800-645-EXPO

YES! PLEASE SEND ME MORE INFORMATION ABOUT MACWORLD EXPO/SAN FRANCISCO! I'M INTERESTED IN:

❑ **Attending** ❑ **Exhibiting** **MT**

Name _____

Title _____

Company _____

Address _____

City/State/Zip _____

Phone _____ Fax _____

email _____

(if you would like to receive information via email about MACWORLD Expo)

Mail to: MACWORLD Expo, 1400 Providence Highway, P.O. Box 9127, Norwood, MA 02062. Or Fax to: 781-440-0357

THIS IS NOT A REGISTRATION FORM.

Owned & Produced by: IDG WORLD EXPO

Sponsored by: Macworld

Managed by: IDG EXPO MANAGEMENT COMPANY

Parsing

A surprisingly difficult task in compiled languages is to allow the user to input a mathematical expression in text form and then evaluate it. This is a trivial one-liner in interpreted languages such as Perl, Python, and Tcl, but C and C++ don't have access to their own parsers and compilers at runtime, so if you want to do this sort of thing you have to write an interpreter yourself. A helpful starting point is the Chipmunk Basic Home Page, which has a large collection of links to source code. In particular, check out interp2num, an expression evaluator which gives you a choice of allowing C-like or BASIC-like syntax for the expressions it evaluates. Also, the STL-Compliant Software Components Collection page has a tokenizer which can help if you want to start from scratch.

Chipmunk Basic Home Page
<http://www.rahul.net/rhn/cbas.page.html>
STL-Compliant Software Components Collection
<http://corp.metabyte.com/~fbp/stl/components.html>

Linear Algebra and Matrix Manipulation

Fortran is famous for high-performance computation, and there a number of well-known libraries for numerical and scientific applications. Although many of these may be available to C and C++ programmers as compiled libraries or by way of the Fortran-to-C compiler, you probably prefer to find true C++ implementations. These allow you to peek under the hood when necessary, and they make it easier to integrate with the rest of your application by providing an object-based interface.

Most of these libraries fall under the general heading of linear algebra, since many problems are naturally expressed in terms of matrices. NIST has developed a library called TNT, the Template Numerical Toolkit, which tackles the difficult problem of developing an efficient yet elegant library for matrix math. One of its design criteria is compatibility with the STL, so that its matrices can act as generic container classes. Oleg Kiselyov's library, called simply LinAlg, is worth a look as well. Even if you do not use it directly, it uses a number of clever programming techniques, such as lazy evaluation and specialized constructors, which are instructive in their own right. Finally, Newmat, by Robert Davies, is a large and well-documented library, and is suitable for working with larger matrices.

Mac F2C
<http://www.alumni.caltech.edu/~igormt/Mac_F2C.html>
TNT Home Page
<http://math.nist.gov/tnt/>
Oleg Kiselyov's LinAlg Library
<http://hyperarchive.lcs.mit.edu/cgi-bin/NewSearch?key=LinAlg>
Newmat09: C++ matrix library
<http://webnz.com/robert/nzc_nm09.html>

(Pseudo) Random Numbers

It's pretty common for programs to use random number generators, and some are more serious about it than others. (Are you designing a solitaire game or trying to predict tornadoes?) If you need something a little more random than rand() or QuickDraw's Random(), there are alternatives out there. It's ironic, if you think about it, that the better implementations call themselves *pseudo*random rather than random, to explicitly acknowledge that you can't really generate truly random numbers on a computer without some extra hardware making quantum measurements. UltraLib is a fast random-number generator, with implementations available in 68K and PPC assembly. It claims to be random even at the bit level, and to have an extremely long period (> 10^356). Agner Fog has another implementation of the same algorithm, as well as an algorithm of his own invention, on his page. Robert Davies, mentioned above, has a random number class library, Newran, which can generate sequences following a number of distributions. Donald E. Knuth's page, also mentioned above, has a public domain random number generator, with code in C or Fortran for single or double precision.

UltraLib
<http://hyperarchive.lcs.mit.edu/cgi-bin/NewSearch?key=RandomNumberLib>
Agner Fog's Pseudo random number generators
<http://announce.com/agner/random/>
Newran02A: C++ random number generator library
<http://webnz.com/robert/nr02doc.htm>
Knuth: Programs
<http://www-cs-staff.stanford.edu/~knuth/programs.html>

Cryptography

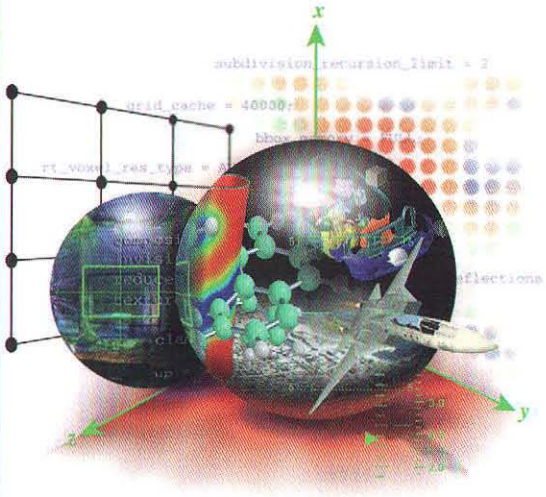
I could certainly write an entire column (or several) on the topic of cryptography, so I will just give a brief pointer here. Bruce Schneier's *Applied Cryptography* is the definitive reference on the subject, and his Counterpane web site has information on two freely available encryption schemes which he invented, as well as a pseudorandom number generator.

Counterpane Homepage
<http://www.counterpane.com/ >

None of these libraries are Mac-specific; some are fully cross-platform, some have Mac versions available, and some may require tweaking (although less tweaking, now that CodeWarrior has support for member templates). In any case, I hope they are useful starting points.

These and armfuls of other links are available from the MacTech Online web pages at <http://www.mactech.com/online/>.

MT



OpenGL™

The Industry's Foundation for High Performance Graphics

Hardware acceleration on every G3 and iMac!

ATI Fully Supports OpenGL

"Conix is an excellent partner in bringing accelerated OpenGL to the Macintosh platform. ATI has been proud of our working relationship with Conix and feel that their modular implementation of OpenGL is elegant in design.... The growing number of 3D Rendering Applicationsand Games being developed using the Conix solution is a testament to Conix's robustness and easy of use."

Henry Nienhuis, ATI Technologies Inc., Manager, Software Engineering

Features

- Geometric primitives (points, lines and polygons)
- RGBA or color index mode
- Viewing and modeling transformations
- Texture Mapping
- Materials Lighting
- Shading
- Z Buffering Atmospheric Effects (fog, smoke, and haze)
- Alpha Blending (transparency)
- Antialiasing
- Accumulation Buffer
- Stencil Planes
- Display list or immediate mode
- Polynomial Evaluators (Supports Non-uniform rational B-splines)
- Feedback, Selection, and Picking
- Raster primitives (bitmaps and pixel rectangles)
- Pixel Operations (storing, transforming, mapping, zooming)
- Gouraud Shading
- Double Buffering
- Transforms

Mac

OpenGL for the Mac OS

An OpenGL SDK for creating native OpenGL applications on the Macintosh. Support for all RAVE and Glide based hardware.

eXodus™

OpenGL for eXodus and X-Ten

Run OpenGL X-Windows applications from your Macintosh under White Pine's eXodus or Tenon's X-Ten. Applications can be run remotely on SGI or other UNIX workstations and render graphics on your Macintosh.

OpenGL for MachTen and CodeBuilder

An OpenGL SDK for creating, modifing and compiling OpenGL code for UNIX on your Macintosh. Works through Tenon's MachTen or CodeBuilder. Complete source builds are included for aux, tk, and glut.

OpenGL for PPCLinux

Conix OpenGL for PPCLinux brings the Conix modular architecture to the PPCLinux platform, including a client-side renderer, a GLX server plugin, and plugin driver support.

conix

Conix Enterprises, Inc.

Check out our Special!

http://www.conix3d.com/MacTech.html • 1.800.577.5505

NEWSBITS

by Jessica Courtney <press_releases@mactech.com>

Snowbound Software Announces Fast TIFF and Document Imaging Applet for Netscape and Internet Explorer

Java Imaging Applet Released

Snowbound Software announced the release of a powerful, new Java imaging applet created from their RasterMaster for Java toolkits. This is Snowbound's latest Java imaging solution for cross platform image viewing and manipulation. Some of the potential for this applet's can be seen at <http://www.snowbnd.com/Java/Applet.htm>.

With Snowbound's RasterMaster for Java component library, any Java developer can create a Java applet that download and views, from the Internet or Intranet, imaging formats such as TIFF, BMP, JPEG, MO:DCA, CALS, and GIF through Netscape(R) and Internet Explorer. Additionally, with the DICOM and JEDMICS support option, two significant imaging market segments — medical imaging and military engineering imaging — are now served with true 100% Java technology.

Snowbound's applet allows the browser client to offer functions such as zoom, rotate, pan, scroll, save, and next page without any installation. Additionally, because Snowbound provides an actual toolkit, their customers can modify the applet to create whatever capabilities they need as well as provide their own look and feel.

Snowbound's RasterMaster for Java toolkit with applet capability is available NOW via the RasterMaster for Java component library. Price starts at $1350 for a RM/View developer's toolkit with the RM/Plus Version priced at $1995. Customized versions are available.
<http://www.snowbnd.com>

VOODOO Version Control Tool Integrates with Text Editor Alpha

UNI SOFTWARE PLUS GmbH announced the availability of an integration between the company's award-winning version control tool VOODOO, and the text editor Alpha.

Like the already existing integrations with CodeWarrior and BBEdit, the new integraton with Alpha offers full access to everyday version control tasks from directly inside the host application.

In order to use VOODOO integrated with Alpha, you will need version 1.8.1 or later of VOODOO <http://www.unisoft.co.at/products/voodoo.html> and version 7.1 or later of Alpha <http://alpha.olm.net/>.

The integration package including installer and documentation can be downloaded from UNI SOFTWARE PLUS <ftp://ftp.unisoft.co.at/pub/voodoo/integration/alpha/> as well as from the author's site <http://www.his.com/~jguyer/Alpha/VOODOO.html>.
<ftp://ftp.his.com/pub/jguyer/Alpha/AlphaVOODOO.sit.bin>

Stone Design Ships CREATE 5.0 For OpenStep and Rhapsody Developer Release 2

Longtime OpenStep and Yellow Box developers, Stone Design, announced that Create 5, the powerful draw studio and web page maker has shipped for OpenStep and Rhapsody Developer Release 2. This release incorporates numerous new features on the much imitated but never duplicated Create.

Create is the all-in-one drawing and web page production application that has been shipping for NeXTSTEP/OpenStep for over 9 years. Create has the power and features that graphic designers need to make professional web pages, flyers, letterheads and much more. The Stone Design web site, which was fabricated in Create, shows off the range and power of this application.

Create lists for $695, but is specially priced to move at $249, and includes the web animation making tool, GIFfun. Also included is a free upgrade to the Mac OS X Server or Yellow Box version when they ship. Previous registered users of Create 4.2 can upgrade for $39. Version 3.x can be upgraded for $149.
<http://www.stone.com/Create_Latest.html>.

Steve Jobs to Deliver Keynote Address at Macworld Expo

IDG Expo Management Company announced that Steve Jobs, the interim CEO for Apple Computer, Inc., will deliver the opening keynote address at Macworld Expo January 5, 1999. Jobs, keynote address will take place from 9:00 a.m. – 10:30 a.m. at the Moscone Convention Center in the Esplanade Ballroom on Jan. 5, 1999. The session is open to registered conference attendees only. Macworld Expo/San Francisco annually brings together established and emerging companies to showcase innovative products and services for the Macintosh community. The Macworld Expo/San Francisco show floor and top-rated conference programs serve as the premiere showcase for products, services and training geared to professional and consumers alike. The Macworld Expo/San Francisco exhibit hall opens Jan. 5 and runs through Jan. 8 at the Moscone Convention Center.

Macworld Expo's multiple conference programs cover beginning and advanced uses of the Mac. The Macworld/Pro sessions offer a technology-intensive professional development series for the Macintosh design, management and networking professionals. The Users Conference covers beginning and advanced uses of applications and techniques in all areas of the Mac OS. Macworld Expo's general conference, begins with pre-conference workshops on January 4 and continues through January 8, 1999. Interested parties can save up to $100 by pre-registering by Dec. 4 through the Macworld Expo website at <http://www.macworldexpo.com> or by calling 1-800-645-EXPO.
<http://www.macworldexpo.com>

Quest Software Adds Toad to Its Suite of Oracle Developer Tools

Quest Software announced the addition of TOAD (the Tool for Oracle Application Developers) to its family of database development and management solutions for Oracle. With tens of thousands of combined users, TOAD and Quest Software's SQL Navigator are the two most widely used graphical environments specifically designed for Oracle Developers. With this acquisition, Quest Software becomes the clear leader in this growing market.

TOAD, formerly a freeware product, has enjoyed rapid growth and support from thousands of Oracle developers around the world. In order to provide a higher level of service to this user base, Quest Software will be offering a low cost support and enhancement plan for users who rely on TOAD every day. A limited-support freeware version of TOAD will continue to be available.
<http://www.quests.com/>

OpenGL for the Macintosh 1.5 (SGI API 1.1) Now Available

Conix Enterprises, Inc. announced the availability of OpenGL for the Mac 1.5. This is the OpenGL the Macintosh graphic's community has been waiting for. With incredible performance gains and improved support for hardware acceleration, Conix provides the ultimate solution for 3D graphics on the Macintosh. ATI Fully Supports OpenGL.

Conix provides affordable distribution licensing for application vendors, and internal distribution agreements for our commercial customers who are developing apps for internal distribution.
<http://www.conix3d.com>

iMac is Now Most Affordable Consumer Computer

Apple Computer, Inc. announced the industry's most aggressive financing program which enables customers to buy an iMac for just $29.99 per month. Apple's financing program features instant in-store credit approval, no payments for 120 days, and one of the lowest interest rates available to consumers anywhere.

Participating Apple resellers nationwide, including CompUSA, are offering instant in-store credit approval using Apple's web-based loan application. Approved buyers can take their iMacs home with no down payment, and with no payments for 120 days.

Apple will be supporting this financing program with television, radio, print and outdoor advertising, as well as with in-store merchandising throughout the retail channel.

Apple also announced some enhancements to iMac. All iMacs now come loaded with Mac OS 8.5, Apple's "must have" upgrade to the Macintosh operating system. All iMacs are also now equipped with ATI's Rage Pro Turbo graphics controller and 6MB of video RAM, for increased graphics performance.

Adobe PageMill 3.0 has been added to the selection of consumer software that comes with iMac, allowing customers to easily create and manage their own Internet web pages.
<http://www.apple.com>

Un-SourceSafe 1.0.1

Michael Amorose has released Un-SourceSafe 1.0.1, a simply Macintosh utility which removes the Visual SourceSafe information added to files by Metrowerks' Visual SourceSafe Client. The utility can remove info from up to 1000 files dropped onto it — even if they are nested 100 levels deep. A special optimized version is also available for G3-based machines. Un-SourceSafe is free and can be downloaded from most public Macintosh ftp utility sites such as InfoMac or UMich.
<http://home.earthlink.net/~amorose/devtools/devtools.htm>

Canadian Software Developer Announces Major Upgrade to VR Authoring Tool

Squamish Media Group, Inc., announced that they have released a major upgrade to their popular software application, soundsaVR 1.0, the first product of its kind available to allow the embedding of directional stereo sound into QuickTime VR using Apple's new 'wired sprite' technology.

soundsaVR 1.1 is a fast and efficient way to add sound to photographic or rendered single node and multi node QTVR movies (1.0 and 2.0), and can import any file in sound formats supported by QT3.0 to enhance the immersive experience of QTVR with realistic directional stereo sound.

Output movies made with soundsaVR are saved using the QuickTime 3.0 movie file format — requiring no special plug-ins — for playback on computers running the Mac OS, Windows 95, or Windows NT.

soundsaVR 1.1 is available now at a limited introductory price of $70.00 (US), and can be purchased through the Squamish Media Group, Inc., web site.
<http://www.smgVR.com>

LightSoft Updates Fantasm to Version 5.30

Fantasm is a powerful Macintosh assembly language development environment capable of translating both 680x0 and PowerPC assembly language. Our design ethos is simplicity with power and hence Fantasm's main aims are to make Macintosh assembly language simple to write, to ensure the code produced is running it's fastest and to provide the most powerful environment for Macintosh assembly language programmers.

Version 5.30 adds full AltiVec™ support along with a few bug fixes and tweaks.

Registered Fantasm 5 users who have not received their update should contact Lightsoft.
<http://www.tau.it/lightsoft>

MT

TIPS & TIDBITS

by Jeff Clites <tips@mactech.com>

WideAdd and WideSubtract Under 68K

If you use the PowerPC "wide arithmetic" routines WideAdd and WideSubtract (say to measure time intervals using the Microseconds() call) then you'll find that your code won't compile for 68K any more. Fortunately, this is easy to fix. The following code declares WideAdd and WideSubtract routines for 68K. They are extremely efficient to use because they generate inline 68K instructions, so using them doesn't even involve a function call.

```
#if __MC68K__

  #pragma parameter WideAddInline(__A0, __A1)
  local void WideAddInline(unsigned long *target_low_word,
                           const unsigned long *source_low_word)
    THREEWORDINLINE(0x2011, 0xD190, 0xD189);
    // move.l (a1),d0  add.l d0,(a0)  addx.l -(a1),-(a0)
  #define WideAdd(target, source) \
          WideAddInline(&(target)->lo, &(source)->lo)

  #pragma parameter WideSubtractInline(__A0, __A1)
  local void WideSubtractInline(unsigned long
    *target_low_word, const unsigned long *source_low_word)
    THREEWORDINLINE(0x2011, 0x9190, 0x9189);
    // move.l (a1),d0  sub.l d0,(a0)  subx.l -(a1),-(a0)
  #define WideSubtract(target, source) \
          WideSubtractInline(&(target)->lo, &(source)->lo)

#endif
```

The three-word 68K inline routines first add the least significant words together (or subtract them), and then do and add with carry (or subtract) on the most significant words to get the final answer. Because 68K byte order stores the low word *last*, the actual routines WideAdd and WideSubtract are declared as macros which first calculate the addresses of the low words of the "Wide" parameters, and then pass those addresses to the inline routines that do the actual work. The end result of this is that you can use WideAdd and WideSubtract just the same as you do in your PPC code, without any source code changes:

```
WideSubtract(&target, &source);
```

Stuart Cheshire <cheshire@cs.stanford.edu>

File Filters Under Navigation Services

There is one subtle, but important, difference between NavGetFile() and its Standard File equivalent—namely that NavGetFile() with a non-nil typeList only shows files matching the specified creator code! For example, if you include 'TEXT' in your typeList, along with your creator code, and expect NavGetFile() to show all text files, you'd be mistaken—you would only see those you created. The only easy way I know of around this is to dispense with the typeList, and do all file selection in your filter routine, something like this:

```
static pascal Boolean FilterTypes
(
  AEDesc                *TheItem,
  void                  *info,
  void                  *callBackUD,
  NavFilterModes        filterMode
)
/* file filter routine for Navigation Services that allows acceptance of all text files,
regardless of creator. */
{
  Boolean               IncludeIt;
  NavFileOrFolderInfo  * TheInfo = (NavFileOrFolderInfo*)info;

  If (theItem->descriptorType != typeFSS){IncludeIt = true;}
  else if (TheInfo->isFolder)           {IncludeIt = true;}
  else
  {
    IncludeIt =
      (TheInfo->fileAndFolder.fileInfo.finderInfo.fdType
            == 'TEXT');
  }

  return IncludeIt;
}
```

Note that the filter routine passes through all folders. My first attempt was mistakenly returning false for all folders, which not only was wrong, it also caused Navigation Services to crash. Also, don't forget to pass the routine to NavGetFile() using a routing descriptor rather than a raw function pointer, and to dispose of this routine descriptor using DisposeRoutineDescriptor() when you are finished with it:

```
NavObjectFilterUPP filterProc =
                    NewNavObjectFilterProc(FilterTypes);
```

Lawrence D'Oliveiro <ldo@geek-central.gen.nz>

MT

Send us your tips or we'll install EvenBetterBusError on your machine! On the other hand, we might just pay you ***$25*** *for each tip we use, or* ***$50*** *for Tip of the Month. You can take your award in goods, subscriptions or US$. Make sure any code compiles, and send tips (and where to mail your winnings) to our* ***Tips e-mail address*** *at tips@mactech.com. (See page 2 for our other addresses.)*

ADVERTISER/ PRODUCT LIST

LIST OF ADVERTISERS

LIST OF PRODUCTS

"...the sharpest trade weekly ever to cross my eyes, containing all the info and ads that interest me."

"Thank you for such a wonderful magazine. I only wish that it were daily because I'm always so anxious to get the next issue."

"Good source of info I can't get elsewhere."

"I usually read it ahead of the stack of magazines I receive each month."

"I appreciate the broad range of topics and the fact that multiple platforms are covered."

"It covers such a broad range of information in a field that requires consistent and frequent coverage."

"I like the fact that you now go more in depth with the other platforms in use in the real world."

"Like the up-to-date info on Mac and other PCs. Helps in decision making."

"I have based several purchases on information from eMediaweekly in the past, and have been rarely disappointed."

"The articles really zero in on what 'I need to know.' I feel as though if I've read this magazine, I am 'informed.'"

""It reflects the new world of cross platform issues and developments..."

"Beautiful prepress and presswork. Smart editing, very informative and useful as a tool around here."

"It is the only pub that I now read for the graphics industry."

"It's excellent."

"My time is at a premium. I get the most current information without wading through the other five monthly pubs I receive."

"...it still seems to be the best overall view of the industries touched by graphic design."

"Only publication of its sort...great guys!"

"...It is by far the de facto source that often determines the direction our business changes."

The Reviews Are In!

We recently asked subscribers what they thought of *eMediaweekly*. In addition to their comments, nearly 90% of all respondents expressed satisfaction with *eMediaweekly*!

Of course we weren't that surprised by the comments, since *eMediaweekly* is expressly designed for the unique information needs of digital media managers involved in purchasing high-end content creation tools. Whether they're publishing on paper or in pixels, digital media managers need to keep abreast of changing technology with the latest news and reviews. That's why they turn to *eMediaweekly*—the ONLY newsweekly covering the *entire* content creation market for print, Web and multimedia—*regardless of platform*.

FREE Subscription!

Subscribe today so you'll have ALL the answers you'll need tomorrow. Go to www.subscribe.emediaweekly.com to get your FREE subscription!

Published by Mac Publishing, L.L.C., a joint venture between Ziff-Davis Publishing Co. and IDG Corp.

«eMediaweekly»

THE NEWSWEEKLY FOR DIGITAL MEDIA MANAGERS